APPROXIMATE SYMBOLIC MODEL CHECKING USING OVERLAPPING PROJECTIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Gaurishankar Govindaraju
August 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun

Approved for the University Committee on Graduate Studies:

To *Amma* and *Nanagaru* ..

Abstract

Bugs in hardware cost money. Whenever an error creeps into a design, time and money must be spent to locate the problem and correct it. With the growing complexity of digital systems, and the tremendous pressure for early-time-to-market schedules, the need for verification tools that can help designers catch bugs at an early stage in the design process cannot be overemphasized.

Traditional methods of verification are empirical in nature and are based on extensive *simulation* of hand-written or automatically generated diagnostic test vectors. Although provably effective in the early stages of the debugging process, their effectiveness drops quickly as the size of the state space grows larger. There has been extensive research on more formal methods based on the use of *theorem provers* to comprehensively verify designs. But these techniques are time consuming and often require a great deal of human expertise to construct a detailed logical proof.

An alternative formal verification approach is *model checking*, in which efficient search procedures are used to automatically determine if the state space of a design satisfies an abstract logical specification. *Symbolic model checking* extends the scope of verification problems that can be handled automatically, by using symbolic representations with binary decision diagrams (BDDs) rather than explicitly searching the entire state space of the model. However, even the most sophisticated symbolic model checking methods cannot be directly applied to many of today's large designs. *Approximate symbolic model checking* is an attempt to trade off accuracy with the capacity to deal with bigger designs. This work explores the idea of using a new approximation scheme called **overlapping projections**.

Under this new approximation scheme, the state space is represented using a

collection of BDDs that constrain possibly overlapping subsets of the state variables in the system. This new scheme is more general and significantly better than earlier approximation schemes.

The ideas are evaluated on publicly available benchmarks from the ISCAS-89 benchmark suite. We report orders of magnitude improvement in the results obtained when compared with earlier schemes. The ideas are also applied to extensively verify a real large design example from the I/O unit of the MAGIC chip in the Stanford FLASH multiprocessor.

Acknowledgments

Interdependence is a higher value than independence. — Stephen R. Covey.

It is indeed hard to do justice here and acknowledge the help and contribution of *all* the people, without whose encouragement and support this thesis would have never seen the light of the day. I can only thank in one blanket statement, all those whom I have had the pleasure of interacting with during the course of my education. You have all enriched my life in so many ways and made it so worthwhile. I will nevertheless attempt to list as many as I can remember. Please forgive any errors of omission. I feel a deep sense of gratitude:

• to my advisors, Prof. David L. Dill and Prof. Mark A. Horowitz. Dave's open-door policy for his graduate students has helped me to sample his opinion and views on the many rough ideas I started with. Dave devotes considerable time and energy to his students, and many of the ideas in this thesis originated from the lengthy brainstorming sessions I had with him. His insistence on continually testing one's ideas on practical, real designs has helped refine this thesis. Finally, his patience, humor, good advice (research and otherwise) and friendship have made my graduate studies at Stanford very enriching and a lot of fun.

• to the other reading committee members: Prof. Mark Horowitz and Prof. Kunle Olukotun, who read drafts of this thesis quickly and carefully. Their suggestions have helped bring more clarity to this dissertation.

• to my parents, for their constant support, encouragement and prayers. Even though there are no formal degrees awarded for the kind of things I learned from them, I

am convinced that the things they taught me have helped me in every step of my education over the years. My brothers, sisters, nephews, nieces and siblings-in-law have always given me the hope and support that made this work possible.

• to people from the FLASH team. In particular I would like to thank Hema Kapadia and Jules P. Bergmann. Hema was the chief designer of the I/O unit in the MAGIC chip. I cannot overemphasize how indebted I am to her for her patience and speedy response to the innumerable questions I have asked her over the years. Jules was the ideal bouncing board to help refine many of the ideas in this thesis. Furthermore, one of the CAD tools that he developed at Stanford, called *vex* [4], was instrumental in helping me to conduct my research.

• to my fellow graduate students from Dave's research group. In particular, Han Yang, Jules Bergmann and I would meet regularly in "core-dump" meetings to thrash out recent papers and ideas.

• to my friends from ASHA Stanford, who are doing very good volunteer work in order to help alleviate the problem of illiteracy in rural India. Interacting with these motivated people has helped me to adopt some of their enthusiasm in other facets of life too.

• to my friends from my tennis league: Craig April, Jay Borenstein, Russ Garber, Victor Lam, Laurent Pierrot... You have all helped make my weekends so much fun here. Tennis has provided balance and well-roundedness to my life here at Stanford as a graduate student.

• to Charlie Orgish and Thoi Nguyen. No research work in the Computer Systems Laboratory at Stanford would ever be possible without the timely help from these super-efficient and capable system administrators.

• to the funding agencies that sponsored this research. This work was supported by DARPA contracts DABT63-94-C-0054, DABT63-96-C-0097 and GSRC contract SA2206-23106PG-2.

• to Deborah Harber for proofreading this thesis. Thanks to her, at least some of the sentences in this thesis are now free of grammatical errors. Any left over grammatical errors are solely my fault.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

> *It is widely agreed that the main obstacle to "help computers help us more" and relegate to these helpful partners even more complex and sensitive tasks is not adequate speed and unsatisfactory raw computing power in the existing machines, but our limited ability to design and implement complex systems with sufficiently high degree of confidence in their correctness under all circumstances.* — Amir Pnueli.

This thesis presents a new approach for the formal verification of digital systems. This chapter motivates the need for verification tools as an aid to the design of digital systems. It provides basic background on existing methods and explains why approximate model checking is an appropriate approach for today's large industrial designs. Finally, there is a note on the scope, the main contributions and the results of this thesis.

1.1 Motivation for Verification Tools

Consider the following interesting developments in the electronics industry:

- Since the first integrated digital circuit, we have witnessed a continuous growth

in the complexity of digital circuits that are designed and fabricated. In particular, Moore's law [55, 56], which states that the number of transistors on a chip will double every eighteen months, has roughly withstood the test of time for the last 35 years.

- In the midst of the growing complexity of these designs, there is also a tremendous pressure to get early time-to-market schedules to maintain business competitiveness.

This leads to the need for tools that can give a quantum jump in designer productivity, enabling him or her to correctly design even more complicated systems in lesser time. In order to enable the designer to work at a much higher level of productivity, the focus of the design effort has moved towards higher abstraction levels. This move is made possible by the introduction of a slew of computer-aided design (CAD) tools that automate the design process at lower levels of abstraction.

Consider the effects of this boom in the electronics industry:

- There is a ubiquitous invasion of hardware systems into our daily lives. Embedded systems inside automobiles, airplanes, cell phones *etc.* are critical parts of our daily lives now.

- With the success of the Internet and embedded systems in general, we can expect our daily lives will be more increasingly dependent on such systems.

A direct consequence of this tight coupling of our daily lives with electronic systems is that it becomes imperative that they function correctly under all scenarios, and their correctness is ensured before they are allowed to be a part of our daily lives. The disastrous effects of having a faulty chip in an embedded system inside an automobile or an airplane are obvious. Such safety critical applications require absolute guarantees of correctness (though in practice, they are a long way from absolute).

Apart from such safety critical applications, electronic goods like computers are slowly becoming a common tool in the average home. Computers and electronics inside home appliances need to be bug-free, or else the financial cost is often prohibitively expensive. For example, the Pentium floating point division bug cost Intel

corporation [43] close to half a *billion* dollars [37]. An earlier detection of the bug would have saved Intel corporation from this huge financial loss and also the associated public relations fiasco. Another less famous example is a disk drive problem, which cost Toshiba corporation [65] nearly a billion dollars. The importance of developing tools that enable the detection and elimination of bugs early in the design cycle cannot be overemphasized.

Thus we see a utopian demand for higher productivity and bug-free designs. Tools that can allow designers to work at higher levels of abstraction take care of the *high productivity* demand. *Verification* tools that can help a designer catch bugs at an early stage in the design process help meet the *bug-free* requirement.

1.2 Verification Methods

Existing methods for verification of digital systems can be classified generally as being empirical or formal. We briefly discuss these methods here.

1.2.1 Empirical Methods

Empirical verification methods attack the problem of design verification by generating and applying tests to a model of the design. The effect of the input tests is then simulated in the model. To simulate a design description, one needs to have rules defining how the statements in the description should be interpreted and executed. Empirical methods do not attempt to *prove* correctness of a design and produce a yes/no answer, but rather to derive a *level of confidence* that the design is free of any obvious errors. The effectiveness of empirical methods depends directly on the effectiveness of the *metrics* used to grade the quality of the tests. Several coverage metrics like line coverage, basic block coverage, toggle coverage, state coverage, tag coverage [25] *etc.* have been proposed, but it is not clear which metric would be most effective in exposing design errors.

Although empirical methods are provably effective in the very early stages of the

debugging process (when the design is still infected with multiple bugs), their effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs. Hence, there is an increasing interest in more formal methods.

1.2.2 Formal Methods

Formal verification methods aim at establishing that an implementation satisfies a specification. The term *implementation* refers to a *model* of the design to be verified, while the term *specification* refers to a more abstract model or a *property* with respect to which the correctness is to be determined. One can ponder endlessly on the philosophical impossibility of proving a system is correct: Is the specification correct? Is the model accurate? Is the verifier correct? Is the computer used to run the verifier correct? Cohn [17] gives a very good analysis of the fundamental obstacles to proving a hardware design correct. However, under the assumption that the *model* is indeed representative of the actual design, and the assumption that the specification is a golden reference model, formal verification techniques can be applied.

A formal model of the underlying design with a precisely defined meaning, enables the application of mathematical proof techniques. This model can be expressed in a variety of mathematical formalisms. Examples of formalisms at the behavioral level are data flow graphs, process algebras and higher order logics, while at the lower levels, finite state machines and switch level models are often used. The design can be modeled directly in one of these formalisms, or a formal model can be constructed from a design description in a hardware description language.

1.3 Formal Verification Methods

Formal verification methods are often divided into the two categories of theorem proving and model checking. We will discuss these in turn. More comprehensive surveys of formal verification methods can be found in [35].

1.3.1 Theorem Proving

Theorem proving (also referred to as *deductive verification*) is an approach to verification where the verification problem is described as a theorem in a formal theory. A formal theory consists of a language in which the formulas are written, a set of axioms and a set of inference rules. The inference rules are syntactic transformation rules for the formulas. With these rules and axioms, theorems can be proved.

However, theorem proving is a time-consuming process that can be performed only by those who are educated in logical reasoning and have considerable expertise. This lack of automation makes its usage rare and limited to guaranteeing the correctness of safety critical systems and protocols.

The advantage of this method is it gives one the ability to reason about infinite state systems, and enables one to check for complex correctness conditions in such large systems. Furthermore, theorem provers also support powerful techniques, such as proof by induction, and they allow the direct verification of parameterized designs without having to instantiate the parameters. However, there is no bound on the time or memory that may be needed to find a proof.

1.3.2 Model Checking and Language Containment

Model checking and language containment are methods to check properties of a design, where the properties are specified respectively as temporal logic formulas [58] and ω-automata [64]. For finite state models, these methods can be fully automated.

However, in practical applications the size of the model often constitutes a major limitation. To keep the size of the model tractable, a compact model is chosen which abstracts away the details that are irrelevant to the property that has to be checked. The selection of a suitable abstraction is typically not automated, because many abstractions only weakly preserve the properties of the design, *i.e.*, if a property is not valid in the abstract model, it can still hold in the exact model. More information on abstraction techniques can be obtained from [14, 49, 20].

A well known academic tool for model checking is SMV [51], developed at Carnegie Mellon University. It supports symbolic model checking of temporal logic formulas.

The term *symbolic* means that the finite state model is not stored explicitly, but instead it is represented as binary decision diagram (BDD) [6]. This is a popular data structure for representing Boolean functions, and is used in many automated verification tools. Another academic tool, which supports both model checking and language containment is VIS [5]. It is a BDD-based environment for design verification, developed at the University of California, Berkeley. Some successful applications on industrial designs [24] have been reported for verification methods based on symbolic model checking.

1.4 Model Checking: Better Choice in Industrial Settings

There are a number of requirements which a formal verification method must meet in order to be valuable in an industrial design environment. Eijk [23] provides a good listing of desirable parameters of a verification tool. An obvious requirement is that the method should be correct. In each of the requirements mentioned below, model checking appears a more appropriate match (compared to theorem proving) in an industrial setting.

- **Automation**

  To minimize the required amount of user guidance, a formal verification method must provide a high degree of automation. Model checking is more amenable to automation than theorem proving, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving.

  Note that the desire to have methods with a high degree of automation does not mean that a tool should not provide options for the user to guide the verification process. Even for fully automated tools, a small amount of user guidance can sometimes result in a significant increase in performance.

- **Error Diagnosis**

When an error is detected in a design description, the verification method must help the designer in locating the error. It should at least be able to produce a pattern of input stimuli which forms a counterexample for the property being verified.

- **Predictable Performance**
  A formal verification method must be able to handle designs of industrial complexity. The keywords are efficiency and predictability. Since formal verification methods are typically computationally extensive, it is difficult to meet the efficiency requirements. The performance of a formal verification tool must degrade gracefully with increasing design size. Small changes in the design should not have a major negative impact on the performance of the verification method. It is also important that the performance is predictable. Before a specific phase of a design project is started, it should be possible to predict if a specific verification method will be able to handle the design or not. In this regard, neither theorem proving nor model checking do well enough, but there is growing evidence [24] of in-house model checkers being developed in most advanced semiconductor processor manufacturing companies.

- **Seamless Integration in the Design Flow**
  To make a verification method convenient to use, it is necessary to tightly integrate it with the design environment. It should be possible to use the same description for simulation and formal verification. A verification method should be able to handle the design styles used in the implementation, and also handle the hardware design description languages used to describe the designs and the cell libraries.

Model checking's support for automation and error diagnosis give it an advantage over theorem proving, at least in an industrial setting.

1.5 The Flow of Model Checking

Applying the model checking technique typically is a three stage process. (For a more detailed analysis on the application of model checking, Clarke *et al.* [15] is an excellent source.)

- **Modeling**: The first task is to convert a design into a formalism accepted and understood by a model checking tool. In many cases, this is simply a compilation task. In other cases, owing to constraints on time and memory, the modeling process may require abstraction to eliminate irrelevant and unimportant details.

- **Specification**: Before verification, it is necessary to state the correctness properties that the design must satisfy. The specification is usually given in some logical formalism. For hardware systems, it is common to use *temporal logic* [58], which asserts how the behavior of the system evolves over time. An important issue in specification is *completeness*. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that are required for the correct functioning of the system.

- **Verification**: The state space of the model is systematically explored. If all the reachable states satisfy the property being checked for, we are done. Otherwise, if any states violating a user defined temporal property are visited, a counterexample trace from the initial states to such error states is presented. After inspecting the counterexample, a designer then makes appropriate modifications to the design, and the model checking exercise is repeated. A final possibility is that the verification task will fail to terminate with a yes/no answer, due to memory size restrictions. In this case, it may be necessary to re-do the verification exercise after changing some of the parameters of the model (*i.e.* by using additional abstractions).

Given the exponential growth rate of the number of states with the number of state variables, the third step in this flow may appear inefficient since it requires exhaustive exploration of the state space. However, with the advent of symbolic

model checking [51], which allows exhaustive *implicit* enumeration of an astronomic number of states, it has completely revolutionized the field of formal verification and transformed it from a purely academic discipline into a practical technique.

1.6 Why Approximate Symbolic Model Checking?

The key data structure used in symbolic model checkers is a BDD (Binary Decision Diagram) [6]. More details on BDDs are in chapter 2; however, for the moment, it suffices to say that it is a data structure for representing Boolean functions.

Binary Decision Diagram (BDD) [6] has proved to be a viable data structure for doing symbolic reachability on large hardware designs. However, for many large design examples, even the most sophisticated BDD-based verification methods cannot produce exact results. This is because the size of the intermediate BDDs, while computing the reachable state space with exact symbolic model checking algorithms, blows up well beyond the memory capabilities of most machines. The blowup of BDDs happens because of the large number of state variables in today's designs, which results in the intermediate BDDs in exact symbolic model checking algorithms that have a large number of variables in their support. As a rough rule of thumb, BDDs with large support sets are more likely to suffer from size explosion problems than those with smaller support sets.

One alternative is to trade accuracy for BDD size requirements, by using approximate verification algorithms. This thesis exploits the key intuition that BDDs with smaller support sets are less likely to blow up. In our approximation scheme, the number of variables in the support of the various BDDs is restricted and controlled so as to keep the BDD sizes well behaved.

More formally, the approximation scheme proposed in this thesis is based on *overlapping projections* of sets of states. A set of states, given by a BDD $S(\mathbf{x})$, is instead represented by a list of BDDs, each element of the list constrains possibly overlapping subsets of the variables in support of S, *i.e.* $\mathbf{x}$. The projection of a set $S(\mathbf{x})$ of bit vectors onto a set of one-bit variables, w_j (where $w_j \subseteq \mathbf{x}$), is the larger set of bit vectors that match some member of S for all the variables in w_j (the

values of the other variables are ignored). $S(\mathbf{x})$ can be approximated by projecting it onto many different subsets of the state variables, and considering S_{ap} to be the intersection of all the projections. The approximation scheme guarantees that the relation $S \subseteq S_{ap}$ holds. Since each of the projections has restricted support from within w_j's (note $w_j \subseteq \mathbf{x}$), they are more robust and less likely to suffer from BDD size blowup problems.

Even though there is some loss of information in any approximation scheme, approximate verification algorithms can often yield useful results. For example:

- Say we are interested in checking if a property holds in every reachable state. Let a BDD S represent the set of reachable states, and S_{ap} be a superset of S. Although S_{ap} is a larger set than S, the BDD for S_{ap} may have a smaller representation, so the computation of S_{ap} may be more efficient than S. If every state in S_{ap} satisfies the property, we can be sure that every state in S also satisfies the property. Hence, a sufficiently accurate approximation can yield a useful result.

- Say we are interested in checking if certain error states are reachable. Let a BDD R represent the set of states that can reach the error states, and R_{ap} represent a superset of R. Once again, even though R_{ap} is a larger set than R, the BDD for R_{ap} may have a smaller representation, so the computation of R_{ap} may be more efficient than R. If none of the initial states are included in R_{ap}, we can safely conclude that the error states are surely unreachable.

The key observation that makes such approximate approaches useful is that any required property of a design rarely relies on *every* implementation detail of the design. Therefore, approximate verification algorithms which retain sufficient information may yield useful results while handling larger designs.

1.7 Scope of the Thesis

This thesis develops and implements a theory for practical automatic verification of synchronous hardware designs. Hardware designs are typically partitioned into

datapath and *control* portions. The *datapath* portion typically involves manipulating and steering data from one end of the chip to the other. The *control* portion decides how and in which direction the data gets steered. The ideas in this thesis are primarily evaluated on the *control* portion of hardware designs. The underlying assumption here is that more often than not, subtle corner-case bugs reside in the intricate control parts of a design rather than the more structured datapath.

In hardware verification, a distinction is usually made between two kinds of properties: *safety* properties and *liveness* properties. A safety property asserts that "nothing bad happens", while a liveness property asserts "something good eventually happens". An example of a safety property is "no more than one agent drives a shared bus at any time". An example of a liveness property is "if an agent wants to drive the bus and waits long enough, he will eventually be granted the bus". This thesis focuses on proving *safety* properties in a hardware design. The conservative over-approximation paradigm of this thesis is amenable for only verifying safety properties.

1.8 Contributions and Results of the Thesis

In short, this thesis presents a new method to compute reasonably accurate over-approximations (superset) of the reachable state space and doing approximate model checking. Specifically:

- *Overlapping Projections: A new approximation paradigm:* Under this paradigm, the state space is represented using a collection of BDDs which constrain possibly *overlapping* subsets of the state variables of the system. This new scheme of approximation is more general and significantly better than earlier approximation schemes which required the use of *disjoint* subsets of the state variables. Allowing even a few overlap bits makes it possible to capture the interaction between finite state machines at a much lower cost (in terms of memory and time) than earlier schemes.

- *Efficient Reachability Operators for Overlapping Projections:* Under this approximation scheme, an efficient algorithm to compute a tight superset of the

set of states that *can be reached* from the initial states is presented. An efficient algorithm to compute a tight superset of the set of states that *can reach* some error states (states that violate a user-defined safety property) is presented.

- *Auxiliary State Variables to get Better Approximations:* Sometimes the critical communication between state machines happens through wires hidden deep inside the combinational logic. There may not be any state variable explicitly capturing the information inside these internal critical wires. The notion of auxiliary state variables is introduced to capture the information embedded in these critical wires. Using these auxiliary state variables along with the usual state variables enables computing tighter supersets of the reachable state space.

- *Automatic Refinement and Counterexample Generation:* In the cases where the model checker is unable to prove the safety property, a simple heuristic to generate counterexamples is presented. In the case where the counterexample cannot be completed in lieu of the approximation, hints are automatically provided on how the choice of subsets can be improved to further refine the approximation, so as to either facilitate a counterexample or a proof of the property.

These ideas were evaluated on publicly available benchmark circuits from the ISCAS-89 suite. We report *orders of magnitude improvement* in the results obtained when compared with the earlier schemes. The ideas are also evaluated on a real large design example from the FLASH [45] Multiprocessor. In particular, the I/O unit of the MAGIC chip in the FLASH Multiprocessor was extensively verified.

1.9 Overview of the Thesis

Chapter 2 introduces some preliminaries on logic manipulation. In particular, it explains BDDs and the constrain operator, both of which are heavily used in this thesis.

Chapter 3 briefly reviews some of the related work in the context of approximations in model checking. A formal analysis of approximations incurred with overlapping

projections is presented, and its place in the world of approximations is elaborated upon.

Chapter 4 describes efficient algorithms to compute approximate successors of a set of states. The quality of approximation is highly dependent on the choice of projections. The heuristic used to guide the choice of projections is presented. The intuitive rationale behind the heuristic is presented. The results obtained by applying this algorithm to compute a superset of the reachable state set of different design examples are presented.

Chapter 5 describes efficient algorithms to compute approximate predecessors of a set of states. The results obtained by applying this algorithm to compute a superset of the states that can reach error states (states violating a user specified safety property) are presented.

Chapter 6 introduces the notion of auxiliary state variables and how they can enable even more refined reachability. The results obtained by applying the technique to various design examples are presented.

Chapter 7 focuses on our scheme of generating counterexamples from the approximations. Since we compute supersets, the approximation ends up with a tube that has a set of paths. All possible counterexample paths must lie inside the tube. Searching for a counterexample in this approximate tube is liable to failure because of the loss of information incurred while choosing the projections. In case of a failure, automatic hints are provided on how the choice of projections can be improved to either facilitate a genuine counterexample or proof of the property.

Finally in chapter 8, we draw some conclusions, present a summary of the thesis and discuss possible future work.

Chapter 2

Preliminaries

Boolean algebra is the basic mathematical tool for reasoning about digital systems. This chapter introduces some basic definitions of Boolean functions. Binary decision diagram (BDD), a popular data structure to represent Boolean functions is introduced. This chapter further elaborates on how BDDs can be used to model digital systems and how they are used to solve simple verification problems. Finally, there is a discussion of the constrain operator. The constrain operator tries to reduce the BDD for a given function relative to another BDD representing the *care set*. BDDs and the constrain operator are heavily used for many of the verification algorithms presented later in this thesis.

2.1 Boolean Functions

In a digital circuit, information is represented in binary form. The two discrete values are denoted as 0 and 1. A Boolean function is an expression. A simple expression consists of one of the constants 0 or 1, or it consists of a variable. More complex expressions can be obtained by negating a simple expression, or by combining two simple expressions with a binary operator. There are three basic binary operators: the logical AND operator, denoted by $\wedge$, the logical OR operator, denoted by $\vee$, and the logical NOT operator, denoted by a $\neg$ or a bar over the negated expression. From these three basic operations, other common operations can be derived. Some

examples are shown in Table 2.1.

Table 2.1: Examples of Boolean functions

Function	Notation	Definition
exclusive-or (XOR)	$f \oplus g$	$(f \wedge \neg g) \vee (\neg f \wedge g)$
equivalence (XNOR)	$f \equiv g$	$(f \wedge g) \vee (\neg f \wedge \neg g)$
implication	$f \rightarrow g$	$\neg f \vee g$

If all the variables in a function are chosen from a set $\mathbf{x}$, then the function is said to be a function over $\mathbf{x}$. The *support* of a function f is the set of all variables occurring in f, and is denoted by $supp(f)$.

Let $\mathcal{B} = \{0,1\}$. A Boolean function is a mapping from $\mathcal{B}^n$ to $\mathcal{B}$, with $n \geq 0$. The positive cofactor of a Boolean function f with respect to a variable a is the function that is obtained by replacing every occurrence of a in f by the constant 1, and is denoted by f_a. Similarly, the negative cofactor of f with respect to a is the function obtained by replacing every occurrence of a by the constant 0, and is denoted by $f_{\bar{a}}$. The following identity holds for every Boolean function:

$$f = (a \wedge f_a) \vee (\neg a \wedge f_{\bar{a}})$$

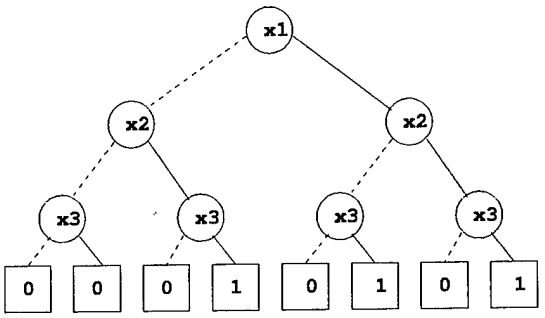
This is known as the Shannon expansion of f, and forms the basis for many BDD based manipulations of Boolean functions. Cofactors are also used to define (Table 2.2) some common operations for quantifiers and substitution.

Table 2.2: Quantifiers and substitution

Function	Notation	Definition
existential quantification	$\exists a \cdot f$	$f_a \vee f_{\bar{a}}$
universal quantification	$\forall a \cdot f$	$f_a \wedge f_{\bar{a}}$
substitution	$f[a \leftarrow g]$	$(g \wedge f_a) \vee (\neg g \wedge f_{\bar{a}})$

x1	x2	x3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth Table
for function f



Decision Tree representing the function f.
(A dashed(solid) branch denotes 0(1) branch)

Figure 2.1: Representing Boolean functions

2.2 Binary Decision Diagrams

Binary Decision Diagram (BDD) is a data structure suitable for representing binary functions. Bryant [6] proposed this representation by imposing restrictions on the representation first introduced by Lee [46] and Akers [2], such that the resulting form is canonical. They are often substantially more compact than traditional representations such as truth tables, conjunctive normal form and disjunctive normal form. Furthermore, they can be manipulated very efficiently. Hence, they have become widely used for a variety of CAD applications.

In particular, BDDs represent a Boolean function as a rooted, directed acyclic graph. As an example, Figure 2.1 illustrates a representation of the Boolean function $f(x_1, x_2, x_3)$ defined by the truth table given on the left, for the special case where the graph is actually a tree. Each nonterminal vertex v is labeled by a variable $var(v)$ and has two children: $else(v)$ (shown as a dashed line) corresponding to the case where the variable is assigned the value 0, and $then(v)$ (shown as a solid line) corresponding to the case where the variable is assigned the value 1. Each terminal vertex is labeled 0 or 1. For a given assignment to the variables, the value yielded by the function is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then

given by the terminal vertex label.

2.2.1 Ordering and Reduction

An *Ordered* BDD (OBDD), has a total ordering $<$ over the set of variables. For any vertex u, and either nonterminal child v of u, their respective variables must be ordered as $var(u) < var(v)$. In the decision tree of Figure 2.1, for example, the variables are ordered $x_1 < x_2 < x_3$.

We further need three transformation rules over these graphs that do not alter the function represented, but result in more compact and canonical representations of the functions.

- **Remove Duplicate Terminals:** Choose a representative terminal vertex for the constant 0 and one representative terminal vertex for the constant 1. All arcs going into a terminal 0 vertex are directed into the representative terminal 0 vertex, and similarly all arcs going into a terminal 1 vertex go to the representative terminal 1 vertex.

- **Remove Duplicate Nonterminals:** If nonterminal vertices u and v have $var(u) = var(v)$, $else(u) = else(v)$, and $then(u) = then(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex. This results in isomorphic subgraphs within the tree being shared. It is this sharing property that enables BDDs be a compact representation for many Boolean functions.

- **Remove Redundant Test:** If nonterminal vertex v has $else(v) = then(v)$, then eliminate v and direct all incoming arcs to $else(v)$.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. We use the term "ROBDD" to refer to a maximal reduced graph that obeys some ordering. For example, Figure 2.2 illustrates the reduction of the decision tree shown in Figure 2.1. Note that on applying the first transformation, the number of terminal nodes are reduced from eight to two, and then the number of nonterminal vertices are reduced by two after the second

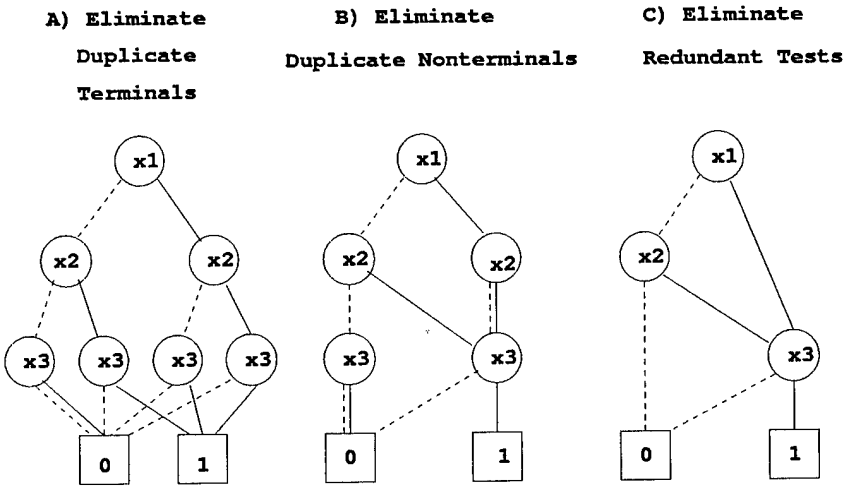


Figure 2.2: Transformations to get ROBDDs

transformation. On application of the third transformation rule another two vertices are eliminated. Since we will always be using this data structure in its ordered and reduced form, unless otherwise mentioned, henceforth we will use the term BDD to mean ROBDDs.

The resulting representation of a function is canonical, *i.e.* for a given ordering, two BDDs for the same function are isomorphic. This property has several important consequences. Functional equivalence can be easily tested. A function is satisfiable iff its BDD representation is not the terminal vertex labeled 0. Any tautological function must have the terminal vertex labeled 1 as its BDD representation. If a function is independent of a variable v, then its BDD representation cannot contain any vertices labeled with v. Thus, once a BDD representation of a function is generated, many functional properties become easily testable.

2.2.2 Effects of Variable Ordering

The form and size of the BDD representing a function depends on the variable ordering. In general, the choice of variable order can make a difference between linear and exponential (in the number of variables) size. For example, Figure 2.3 shows two BDD representations of the Boolean function $f = a_1 \wedge b_1 \vee a_2 \wedge b_2 \vee a_3 \wedge b_3$. The choice of variable order $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$ yields a BDD with 8 vertices,

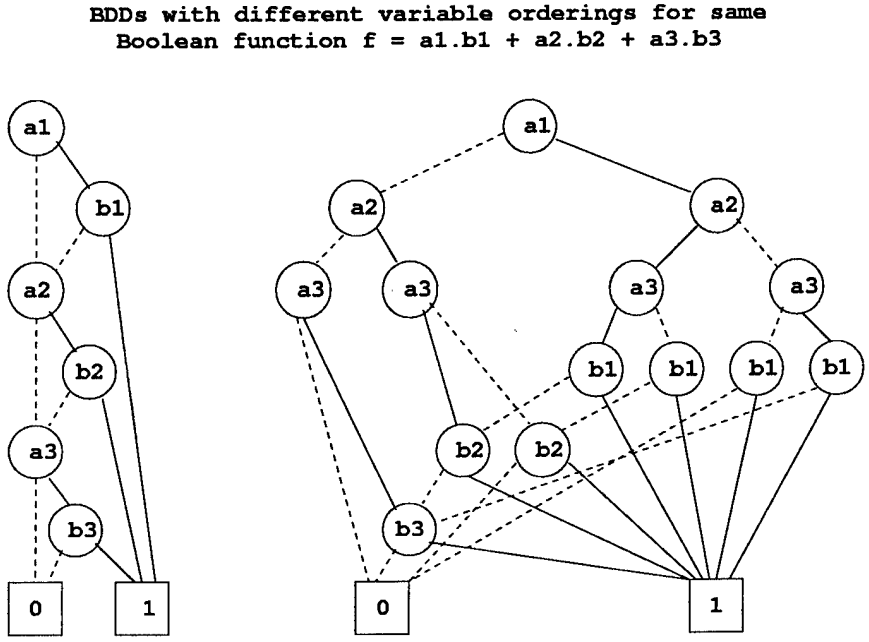


Figure 2.3: Effects of variable ordering

while the choice of variable order $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$ yields a BDD with 16 nodes.

The difference of a factor of two in the previous example may not appear all that dramatic. However, for the more general case of $f = a_1 \wedge b_1 \vee a_2 \wedge b_2 \vee \ldots \vee a_n \wedge b_n$, it can be proved that the first variable ordering $a_1 < b_1 < a_2 < b_2 < \ldots < a_n < b_n$ yields a BDD with $2(n+1)$ vertices, whereas the other choice of variable ordering $a_1 < a_2 < \ldots < a_n < b_1 < \ldots < b_n$ yields a BDD with 2^{n+1} vertices. For large values of n, the difference between the linear growth of the first order and the exponential growth of the second has a dramatic effect on the memory requirements and the efficiency of the manipulation algorithms.

Most applications using BDDs choose some ordering at the beginning and construct graphs for all relevant functions according to this ordering. This ordering is chosen manually or according to some heuristic guided analysis of the underlying functions in the design. For example, several heuristic methods have been devised that, given a logic gate network, often derive a good ordering for variables representing the primary inputs [27, 50]. Note that these heuristics do not need to find the best

possible ordering. As long as an ordering can be found that avoids an exponential growth, operations on BDDs remain reasonably efficient.

Table 2.3: Complexity of BDD algorithms

Operation	Notation	Complexity
Negation	$\neg f$	$O(1)$
Conjunction	$f \wedge g$	$O(\|f\|\|g\|)$
Disjunction	$f \vee g$	$O(\|f\|\|g\|)$
Exclusive-OR	$f \oplus g$	$O(\|f\|\|g\|)$
Equivalence	$f \equiv g$	$O(\|f\|\|g\|)$
If-then-else	$ite(f,g,h)$	$O(\|f\|\|g\|\|h\|)$
Cofactoring	$f_a, f_{\bar{a}}$	$O(\|f\|)$
Existential Quantification	$\exists a \cdot f$	$O(\|f\|^2)$
Universal Quantification	$\forall a \cdot f$	$O(\|f\|^2)$
Substitution	$f[a \leftarrow g]$	$O(\|f\|^2\|g\|)$

Bryant [6] gives algorithms for computing the BDD representations of $\neg f$ and $f \vee g$, given the BDDs for functions f and g. These algorithms have complexity linear in the sizes of the argument BDDs. Table 2.3 gives a brief overview of the complexity of some BDD manipulation algorithms. For example, given the BDDs for the argument functions f and g, the complexity of the algorithm to generate the BDD for $f \vee g$ is proportional to the product of the sizes of the individual BDDs. (We use the notation $|f|$ to denote the number of nodes or the size of the BDD representing the function f.)

Another useful operation over BDDs is quantification over Boolean variables and substitution of variable names. Bryant gives an algorithm for computing the BDD for a restricted function of the form f_a and $f_{\bar{a}}$, *i.e.*, f with the variables a set to 1 or 0. The restriction algorithm allows us to compute the BDD for the formula $\exists a \cdot f$, where a is a Boolean variable and f is a function, as $f_a \vee f_{\bar{a}}$. The substitution of a variable w for a variable v in a function f, denoted by $f[v \leftarrow w]$, can be accomplished using quantification:

$$f[v \leftarrow w] = \exists v \cdot [(v \equiv w) \wedge f].$$

More efficient algorithms [6, 7] exist for the case of quantification over multiple variables or multiple renamings.

2.2.3 Intuition on BDD Variable Ordering

BDDs provide a practical approach to symbolic Boolean manipulation of large designs only when the graph sizes remain well below the worst case of being exponential in the number of variables. As the previous example shows, BDD sizes for some functions are sensitive to the variable ordering chosen but remain quite compact as long as a good ordering is chosen. Further, there is ample empirical evidence indicating that many functions encountered in real applications can be represented efficiently as BDDs. Here, we list some simple yet powerful rules of thumb that can be employed to keep BDD sizes well behaved.

- **Control before Data:** Variables that decide functions earlier should be higher up in the variable order. In particular, in the global variable order, control signals which decide where data gets steered should be above the data bits they steer. As an example, consider a multiplexer, with $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ as data inputs on two input streams, $z_1, \ldots, z_n$ is the output stream and *sel* as the select line, then the control signal *sel* ought to be above the data bit variables in the variable order. This will ensure that BDD for the output of the multiplexer is compact.

  Another common practice among designers is to have a *reset* control signal that causes all the individual finite state machines to transition from whatever state they are to their respective *idle* states. Keeping the variable *reset* up in the variable order helps have compact BDDs for the next state functions of the finite state machines.

- **Interleaved ordering for special hardware:** Comparators, equality detectors and adders are common in digital designs. In each of these cases it is important to keep the bits interleaved. For example, consider a comparator with $x_1, \ldots, x_n$ (x_n is most significant bit) and $y_1, \ldots, y_n$ as data inputs on two input

streams, and *out* as output. In order to ensure that the BDD for the output *out* is compact, it is important to have bits from the two input channels interleaved, *i.e.* the variable order should obey $x_n < y_n < x_{n-1} < y_{n-1} < \ldots < x_1 < y_1$. Note that since comparators compare the most significant bits first, we put them on top of the variable order. So, choosing the other interleaved order $x_1 < y_1 < x_2 < y_2 < \ldots < x_n < y_n$ would yield poor results. However, in case of an equality detector, there is no special significance to the most significant bit, and either of the two interleaved orders would be optimal.

We can consider each output of an n-bit adder as a Boolean function over variables $x_1, \ldots, x_n$, representing one operand, and $y_1, \ldots, y_n$ representing the other operand. The function for any output bit of the adder has OBDD representation of linear complexity for the interleaved ordering $x_1 < y_1 < x_2 < y_2 < \ldots < x_n < y_n$.

Even when these simple rules of thumb are unable to avoid BDD size blowup, there are dynamic variable ordering heuristics [48] which can be invoked automatically once the BDD sizes reach some limit. This typically slows down the manipulation algorithms since improving the ordering is a computationally intensive task. In our experience, the best approach is to start with a manually fixed ordering based on some high level intuition about the underlying circuit, and to enable dynamic ordering in case the initial choice fails in the first pass. The final ordering decided by dynamic ordering is dumped out to a file and used for subsequent simulations. In this way, the computationally intensive cost of dynamic variable ordering is incurred only once and amortized over many simulation runs.

2.3 Modeling Synchronous Hardware with BDDs

We analyze synchronous hardware by modeling it as a Mealy machine. A Mealy machine for our applications is a 4-tuple, and is given as $M = \langle \mathbf{x}, \mathbf{y}, q_0, \mathbf{n} \rangle$, where $\mathbf{x} = \{x_1, \ldots, x_k\}$ is the set of state variables, and $\mathbf{y} = \{y_1, \ldots, y_l\}$ is the set of input signals. We will use $\mathbf{x}' = \{x'_1, \ldots, x'_k\}$ to denote the next state versions of the corresponding

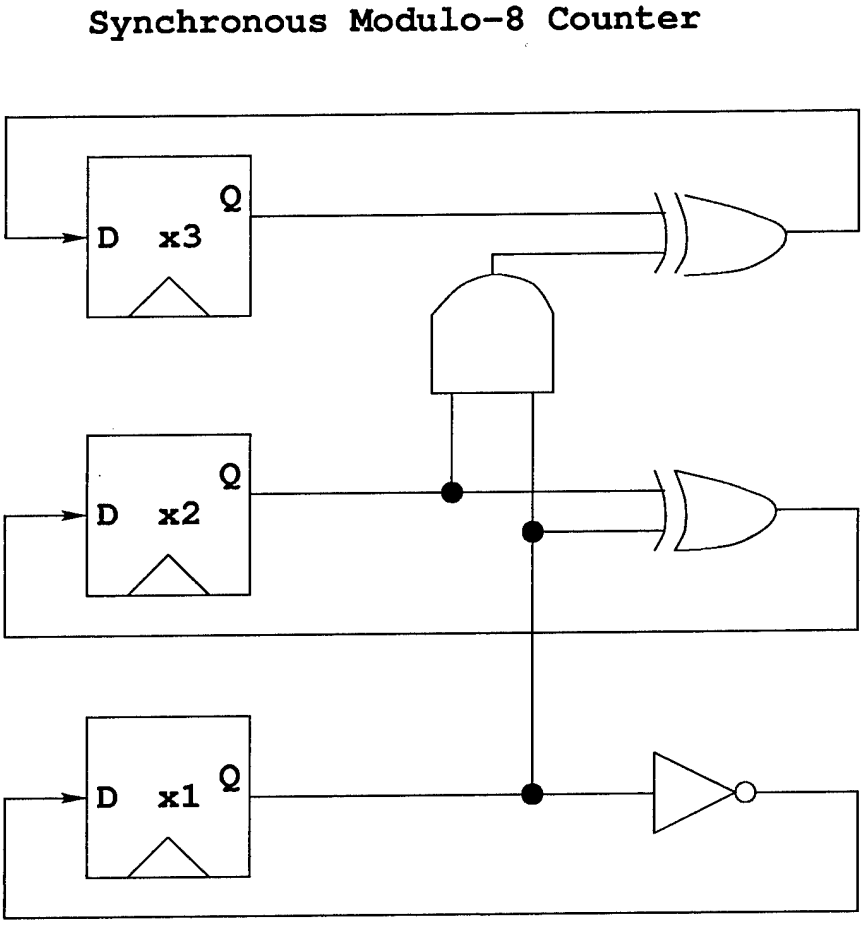


Figure 2.4: Modeling a synchronous circuit with BDDs

variables in $\mathbf{x} = \{x_1, \ldots, x_k\}$. The set of possible states is $\{0,1\}^k$, and the input space is $\{0,1\}^l$. The initial state, q_0, is some state from the state space $\{0,1\}^k$. The next state function vector is $\mathbf{n} : [n_1, \ldots, n_k]$, where the function $n_i : \{0,1\}^k \times \{0,1\}^l \rightarrow \{0,1\}^k$, is the next state function of state variable x_i. (Conventional definitions of Mealy machines include outputs too, but they are not relevant in our applications, since we are only concerned with exploring the state space of the machine.)

Since BDDs can be used to represent functions over finite domains (like Boolean functions) and finite sets, they can be used to represent the next state function $\mathbf{n}$ and the initial states q_0.

This method of modeling a synchronous circuit can be illustrated by using an example. The circuit in Figure 2.4 is a modulo-8 counter. Let $\mathbf{x} = \{x_1, x_2, x_3\}$ be the set of state variables for this circuit, and let $\mathbf{x}' = \{x_1', x_2', x_3'\}$ be the next state versions of these state variables. The next state functions $\mathbf{n} = \{n_1, n_2, n_3\}$ for the state variables are given by

$$\begin{aligned} n_1 &= \neg x_1, \\ n_2 &= x_1 \oplus x_2, \\ n_3 &= (x_1 \wedge x_2) \oplus x_3. \end{aligned}$$

BDDs for these functions can be easily created. Assuming the initial state corresponds to all the state variables being 0, we get a BDD for q_0 by creating a BDD for the function $\neg x_1 \wedge \neg x_2 \wedge \neg x_3$.

BDDs can be used to represent not just sets of states, but also sets of ordered pairs of states. This enables modeling the transitions of a circuit as BDDs. This is done by using the set of state variables $\mathbf{x} = \{x_1, \ldots, x_k\}$ and their corresponding next state versions $\mathbf{x}' = \{x_1', \ldots, x_k'\}$. A valuation for the variables in $\mathbf{x}$ and $\mathbf{x}'$ can be viewed as designating an ordered pair of states in the circuit, and we can represent sets of these valuations using BDDs. Such sets of pairs of states can be used to model the next state transition *relation*. If T is a transition relation, then we use $T(\mathbf{x}, \mathbf{x}')$ to denote the BDD that represents it. In the example of the synchronous modulo-8 counter (Figure 2.4), the individual next state functions for state variable x_i is turned

into a relation t_i as follows:

$$\begin{aligned} t_1(\mathbf{x}, x_1') &= (x_1' \equiv \neg x_1), \\ t_2(\mathbf{x}, x_2') &= (x_2' \equiv x_1 \oplus x_2), \\ t_3(\mathbf{x}, x_3') &= (x_3' \equiv (x_1 \wedge x_2) \oplus x_3). \end{aligned}$$

The individual t_i relations describe the constraints that each x_i' must satisfy in a legal transition. These constraints can be combined by taking their conjunction (for asynchronous circuits we would take the disjunction) to form the transition relation

$$T(\mathbf{x}, \mathbf{x}') = t_1(\mathbf{x}, x_1') \wedge t_2(\mathbf{x}, x_2') \wedge t_3(\mathbf{x}, x_3').$$

In the general case of a synchronous circuit with state holding elements $\mathbf{x} = \{x_1, \ldots, x_k\}$ and associated next state functions $\mathbf{n} = \{n_1, \ldots, n_k\}$, the individual relations are defined as

$$t_i(\mathbf{x}, x_i') = (x_i' \equiv n_i).$$

Continuing the analogy with the synchronous modulo-8 counter, the conjunction of these individual relations forms the transition relation of the whole circuit

$$T(\mathbf{x}, \mathbf{x}') = t_1(\mathbf{x}, x_1') \wedge t_2(\mathbf{x}, x_2') \wedge \ldots \wedge t_k(\mathbf{x}, x_k').$$

Given a BDD for the individual functions n_i, it is straightforward to compute the BDD that represents the transition relation T. Such a transition relation is called *monolithic*, because it is represented by a single BDD.

2.4 Symbolic Reachability Algorithms

Given the BDDs for the initial state and the next state functions of a digital system, we can use standard BDD based symbolic algorithms to compute one step successors (also referred to as image) and one step predecessors (also referred to as pre-image) of

any set of states. These can be done repeatedly to compute all the reachable states.

In our applications, sets can be viewed as predicates, since we can form the characteristic function corresponding to a set. BDDs can be used to represent predicates and manipulate them [7]. For example, let $R(\mathbf{x})$ be a BDD with support in $\mathbf{x}$, we can compute the image of R under $\mathbf{n}$ as

$$Im(R(\mathbf{x}), \mathbf{n}(\mathbf{x}, \mathbf{y})) = \lambda \mathbf{x}'.\exists \mathbf{x}, \mathbf{y}.(\mathbf{x}' = \mathbf{n}(\mathbf{x}, \mathbf{y})) \wedge R(\mathbf{x}).$$

Im produces a predicate with support in $\mathbf{x}'$. The resulting predicate is 1, if and only if $\mathbf{x}'$ is in the image of R under $\mathbf{n}$. The set of reachable states in M can be computed by a least fix point iteration [7]:

$$FwdReach(q_0) = \text{ lfp } R.\lambda \mathbf{x}.(q_0(\mathbf{x}) \vee Im(R(\mathbf{x}), \mathbf{n}(\mathbf{x}, \mathbf{y}))).$$

The notation lfp (least fixed point) is short for the following piece of pseudo-code.

$R_{reached} \leftarrow q_0$
$R_{previous} \leftarrow 0$
while $R_{reached} \neq R_{previous}$ **do**
  $R_{previous} \leftarrow R_{reached}$
  $R_{reached} \leftarrow q_0 \vee Im(R_{previous}, \mathbf{n})$
return $R_{reached}$

(For a very simple tutorial on symbolic model checking, please refer to the Appendix of this chapter in Section 2.6.1.)

Let g be the set of states that satisfies a user specified property, and let $g(\mathbf{x})$ be the BDD representing it. Then the pre-image of $\neg g(\mathbf{x})$, *i.e.* the set of states that can reach a state violating the property g in one step, can be computed as follows:

$$Pre(\neg g, \mathbf{n}) = \lambda \mathbf{x}.\exists \mathbf{x}', \mathbf{y}.(\mathbf{x}' = \mathbf{n}(\mathbf{x}, \mathbf{y})) \wedge \neg g(\mathbf{x}').$$

Pre produces a predicate with support in $\mathbf{x}$. The resulting predicate is 1, if and only if $\mathbf{x}$ is in the pre-image of $\neg g$ under $\mathbf{n}$. The set of states that can reach $\neg g$ in machine

M can be computed by a least fix point iteration [7]:

$$BackReach(\neg g) = \text{ lfp } R.\lambda \mathbf{x}.(\neg g(\mathbf{x}) \vee Pre(R(\mathbf{x}), \mathbf{n}(\mathbf{x}, \mathbf{y}))).$$

As before, the notation lfp, is now short for the following piece of pseudo-code.

$R_{reached} \leftarrow \neg g(\mathbf{x})$
$R_{previous} \leftarrow 0$
while $R_{reached} \neq R_{previous}$ **do**
  $R_{previous} \leftarrow R_{reached}$
  $R_{reached} \leftarrow \neg g(\mathbf{x}) \vee Pre(R_{previous}, \mathbf{n})$
return $R_{reached}$

2.5 Constrain Operator

BDDs are the standard representation of Boolean functions in logic synthesis and formal verification. In both logic synthesis and formal verification, one frequently has don't-care information, which should be used to improve the quality of the solution, the efficiency of the algorithm, or both. Thus, an operator that simplifies a BDD using don't-care information is obviously important. For historical reasons, this problem is usually phrased in terms of a care-set: Given BDDs f and c, find another BDD g (which is hopefully smaller than the BDD for f) that agrees with f for all truth assignments that satisfy c.

Several simplification operators have been proposed. The earliest are the constrain operator (also called the generalized cofactor [18]), denoted by $\downarrow$, and the restrict operator, denoted by $\Downarrow$. Both these operators were proposed by Coudert and Madre [18]. For our applications, constrain is a better match and hence we define it. (More details on restrict can be obtained from [18].)

2.5.1 Definition of Constrain

For a function f and a given care set c, the function $f \downarrow c$ can be interpreted as an incompletely specified function whose onset is $f \wedge c$ and don't care set is $\neg c$.

The generalized cofactor (function definition given below) depends on the variable ordering used in the BDD representation. (If c is a cube, the generalized cofactor is equal to the usual cofactor and is independent of the variable ordering.)

constrain $f \downarrow c$
  assert $(c \neq 0)$
  if $((c == 1)||(f == 0)||(f == 1))$ **return** f
  if $(f == c)$ **return** 1
  if $(f == \neg c)$ **return** 0
  let x_1 *be top variable in* c
  if $(c_{\bar{x_1}} == 0)$ **return** $f_{x_1} \downarrow c_{x_1}$
  if $(c_{x_1} == 0)$ **return** $f_{\bar{x_1}} \downarrow c_{\bar{x_1}}$
  else return $(ite\ \ x_1\ \ f_{x_1} \downarrow c_{x_1}\ \ f_{\bar{x_1}} \downarrow c_{\bar{x_1}})$

2.5.2 Properties of Constrain

Some interesting properties of constrain which we will exploit later in this thesis are listed below. The proofs for these properties are in [52, 66].

- $(f \downarrow f) = 1$, and $(f \downarrow \neg f) = 0$
- $(f \wedge g) \downarrow h = (f \downarrow h) \wedge (g \downarrow h)$
- $(f \downarrow g) \wedge g = f \wedge g$
- $(f \downarrow g) \downarrow g = f \downarrow g$
- if f and h have independent support $f \downarrow h = f$
- $(f \downarrow g = 1)$ *iff* $(g \rightarrow f)$
- $(f \downarrow g = 0)$ *iff* $(g \rightarrow \neg f)$

The belief that constrain never increases the size of the BDD to which it is applied is a common misconception. There exist functions f and g such that the resulting

BDD for $f \downarrow g$ is bigger than the BDD for f. The idea is we choose a function f for which there is a lot of subgraph sharing and then we let constrain destroy some of the sharing.

Example 1 [1] *Consider the family of function* $f_n = x_1 \oplus \ldots \oplus x_n$ *and* $c_n = x_1 \vee \ldots x_n$. *The variable order is irrelevant because of symmetry. For plain BDDs,* f_n *has* $2n+1$ *nodes (including the terminal nodes), whereas* $f_n \downarrow c_n$ *has* $3n-2$ *nodes. For BDDs with complement edges,* f_n *had* $n+1$ *nodes, whereas* $f_n \downarrow c_n$ *has* $2n-1$ *nodes. In either case, the BDD size increases after the constrain operation.*

2.6 Appendix

2.6.1 A Simple Tutorial on Symbolic Model Checking

The key data structure used in symbolic model checkers is a BDD (Binary Decision Diagram) [6]. Computing the set of reachable states using BDDs requires three basic ideas:

- Representing sets of states using BDDs
- Computing successors of sets of states
- Fix point iteration

The first idea is to represent sets of states using BDDs. Basically, we can think of a BDD as representing a set of truth assignments: if the function the BDD represents is true for a given truth assignment, that assignment is in the set; if the function is false, that truth assignment is not in the set. For example, if we consider three Boolean variables x_1, x_2, and x_3, the BDD for the function $x_1 \wedge \neg x_2 \wedge \neg x_3$ represents the set containing only one truth assignment {100}; the BDD for the function $x_1 \vee x_2$ represents the set of six truth assignments {100, 101, 110, 111, 010, 011}, and the BDD for 1 (the Boolean value *True*) represents the set of all eight truth assignments. If we associate a Boolean variable with each flip-flop (state holding element) in a

[1] This example was suggested by Jerry Burch and is also in Hu's thesis [41].

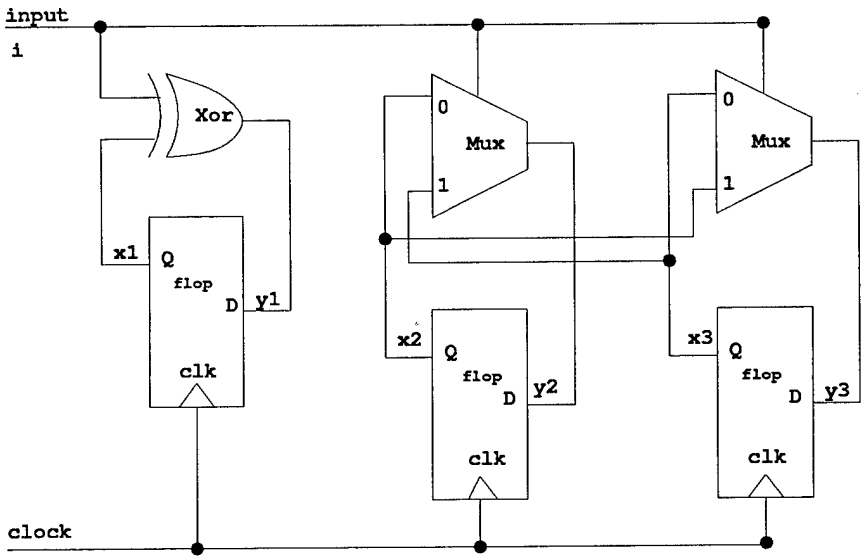


Figure 2.5: Simple finite state machine

circuit, then these BDDs can be viewed as representing sets of states of the state machine.

The next concept is image computation. Basically, if we have a BDD that represents a set of states of a state machine, the image of that BDD is a new BDD that represents the set of states that the machine could be in (assuming a totally nondeterministic assignment to the inputs from the environment) exactly one clock tick later. For example, consider the state machine in Figure 2.5. (The example used in this simple tutorial is from Hu [42].) The BDD for $\neg x_1 \wedge x_2 \wedge \neg x_3$ represents the single state where the flops x_1, x_2 and x_3 are outputting 0, 1 and 0 respectively. Depending on the value of the input, the machine has two possible states at the next clock tick, so the image of this BDD is the BDD for $(\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3)$. The simplest way to compute images is to first build a BDD that represents the relationship between the present and next state values of the flops. This BDD is called the transition relation. In our example, it would be the BDD for $(y_1 \equiv (x_1 \oplus i)) \wedge (y_2 \equiv (\neg i \wedge x_2) \vee (i \wedge x_3)) \wedge (y_3 \equiv (\neg i \wedge x_3) \vee (i \wedge x_2))$. Next, AND the transition relation with the BDD whose image is desired. Then, existentially quantify out the variables for the present state and the primary inputs.

The final idea is an iteration using images to compute all reachable states. It can be evaluated naively by the following pseudo-algorithm:

$$\begin{aligned} R_0 &= \textit{BDD for initial state} \\ R_1 &= R_0 \vee \textit{Image}(R_0) \\ &\vdots \\ R_{i+1} &= R_i \vee \textit{Image}(R_i) \end{aligned}$$

Intuitively, R_i is the set of all states reachable in i or fewer clock cycles from the initial state. For *finite* state machines, this sequence will converge eventually, when $R_{i+1} = R_i$ (which is easy to test, since BDDs are canonical). In this example, the initial state $R_0 = \neg x_1 \wedge x_2 \wedge \neg x_3$, after one iteration $R_1 = (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3)$, and after two iterations $R_2 = R_1$, so we are done.

Chapter 3

Approximation by Overlapping Projections

> *It is the mark of an educated mind to rest satisfied with the degree of precision which the nature of the subject admits and not to seek exactness where only an approximation is possible.* — Aristotle.

The value of approximate methods is that they allow model checking techniques to be applied to larger designs. In this chapter this underlying motivation for approximate methods is brought out. Further, a new scheme of approximation, called *overlapping projections*, is defined. This approximation scheme captures some important interaction between different finite state machines. Finally, this scheme is compared with earlier approximation schemes.

3.1 Why Approximate Methods?

In Chapter 2, we saw how BDDs can be used to compute and represent the exact set of reachable states for finite state machines. Once this exact model is computed, we can very efficiently prove safety properties [51, 58]. However, in practice, the naive algorithms from Section 2.4 are not directly applicable to today's large designs. This

is because of two problems:

- *State Explosion Problem*: Hardware systems are composed of many different components which work concurrently. Each of these individual components may have small manageable finite state machines. However, the global finite state model of the whole concurrent system grows exponentially in size as the number of components in the systems increases. This is widely known as the *state explosion* problem. For large industrial designs it is therefore not uncommon to have designs with more than 10^{1000} states. Thus, methods which attempt to explicitly enumerate the reachable state space one state at a time are very unlikely to succeed.

- *BDD Size Blowup Problem*: In the case of BDDs, there is no direct correlation between the size of the BDD and the size of the set it represents. There are cases where the BDD for a very large set of states is compact, and there are also cases where the BDD for a very small set of states is huge. Empirically, BDDs seem to work well for designs whose size is within 100 state variables. For such designs, the exact model can be computed and represented with BDDs. However, today's designs have thousands of state variables, and a direct application of BDDs to compute and represent the exact model of such designs will usually fail. The intermediate BDDs would need to store functions with thousands of variables in their support, and even with dynamic variable ordering heuristics the size of the BDDs grows extremely big, stretching the memory available to the limit.

Given the hopelessness of exact methods in dealing with large designs, we are forced to settle for approximate methods which trade off accuracy for the capacity to deal with larger designs. However, the situation is not as dim as it seems. In fact, approximate methods often yield a lot of useful information. What makes them useful is:

- *Localized domain of a property*: Any method of approximation involves some loss of information. The key question is to regulate the loss of information so that useful information is still retained by the approximate method. Suppose we are interested in checking if a design satisfies a certain required property. Now, proof

of any single property very *rarely* relies on *every* implementation detail. Hence, if the scheme of approximation can retain sufficient design details relevant to the property being proved, then the approximate analysis can still yield useful results. This *localized effect* of a property renders approximate analysis useful. The *cone-of-influence* [52] reduction is an example of how irrelevant details (relative to a property) from a design can be abstracted away and the remaining simpler design (which is bisimulation equivalent [13]) analyzed instead.

- *Property preservation between different models*: Approximation methods reduce the verification of a system property to the verification of a related property over a simpler system. This enables us to do system analysis in one domain and carry over the results to another domain. Some approximation (or abstraction) is implicitly done when we model a real world system by some mathematical model. In cases where the underlying mathematical model is complex, we can analyze a usually simpler mathematical model. Typical example usage of abstraction is when verification of infinite state systems is done by constructing a finite state abstract system that can be model checked. Abstraction can also mitigate the state explosion problem in the finite state case, by constructing an abstract system with a more manageable set of states.

  We are interested in abstraction mechanisms that allow us to prove properties in the simpler mathematical model and conclude that the more concrete model has some related properties. Property preservation between the two models can be formally justified by showing a formal relation between the two models, using the theory of abstract interpretation. The theory of abstract interpretation, also referred to as *Galois connections* [19], relates the semantics of systems in two different domains. The theory was introduced by Cousot and Cousot [19] and it is still being used in many different settings, ranging from compiler optimization to language semantic analysis, formal verification and theorem proving. (More details on Galois connections, and how our scheme of approximation fits in the Galois connections framework is in the Appendix of this chapter.)

3.2 Approximation by Overlapping Projections

This section introduces a new scheme of approximation called overlapping projections, and forms the basis of this thesis.

3.2.1 Definitions and Theory

Recall from Section 2.3, that $\mathbf{x} = \{x_1, \ldots, x_k\}$ is the set of state variables. Let $\mathbf{w} = (w_1, \ldots, w_p)$ be a collection of not necessarily disjoint subsets of $\mathbf{x}$. (Each subset will also be referred to as a *block*).

Projections and Concretization: We define the operator $\alpha_j(R)$ which projects a predicate $R(\mathbf{x})$ onto the variables in w_j. Intuitively, $\alpha_j(R)$ represents the set of Boolean vectors that agree for the variables in w_j with some Boolean vector satisfying R. Let z consist of all of the Boolean variables in $\mathbf{x}$ that are *not* in w_j. We can define α_j as

$$\alpha_j(R(z, w_j)) = \lambda w_j . \exists z . R(z, w_j).$$

Clearly, the set of Boolean vectors satisfying R is a subset of those satisfying $\alpha_j(R)$. This can be written using logical implication as $R \rightarrow \alpha_j(R)$. The projection operator, α, projects a predicate $R(\mathbf{x})$ onto the various w_j's, and its associated concretization operator γ conjoins the collection of projections. Figure 3.1 is a geometric interpretation of $\langle \alpha, \gamma \rangle$ pair of functions.

$$\begin{aligned} \alpha(R(\mathbf{x})) &= (\alpha_1(R), \ldots, \alpha_p(R)). \\ \gamma(R_1, \ldots, R_p) &= \bigwedge_{j=1}^{p} R_j. \end{aligned}$$

Example 2 *Consider a design with 4 state variables,* $\mathbf{x} = \{x_1, \ldots, x_4\}$. *Let the collection of choice of subsets be* $\mathbf{w} = (w_1, w_2)$, *where* $w_1 = \{x_1, x_2, x_3\}$ *and* $w_2 = \{x_2, x_3, x_4\}$. *(Note that there is some overlap in the sets* w_1 *and* w_2.*) Let* $R(\mathbf{x})$ *be the* one_hot *function, which is true if and only if one and exactly one of the variables in* $\{x_1, \ldots, x_4\}$ *is true. Thus* $R(\mathbf{x}) = x_1\bar{x}_2\bar{x}_3\bar{x}_4 \vee \bar{x}_1x_2\bar{x}_3\bar{x}_4 \vee \bar{x}_1\bar{x}_2x_3\bar{x}_4 \vee \bar{x}_1\bar{x}_2\bar{x}_3x_4$.

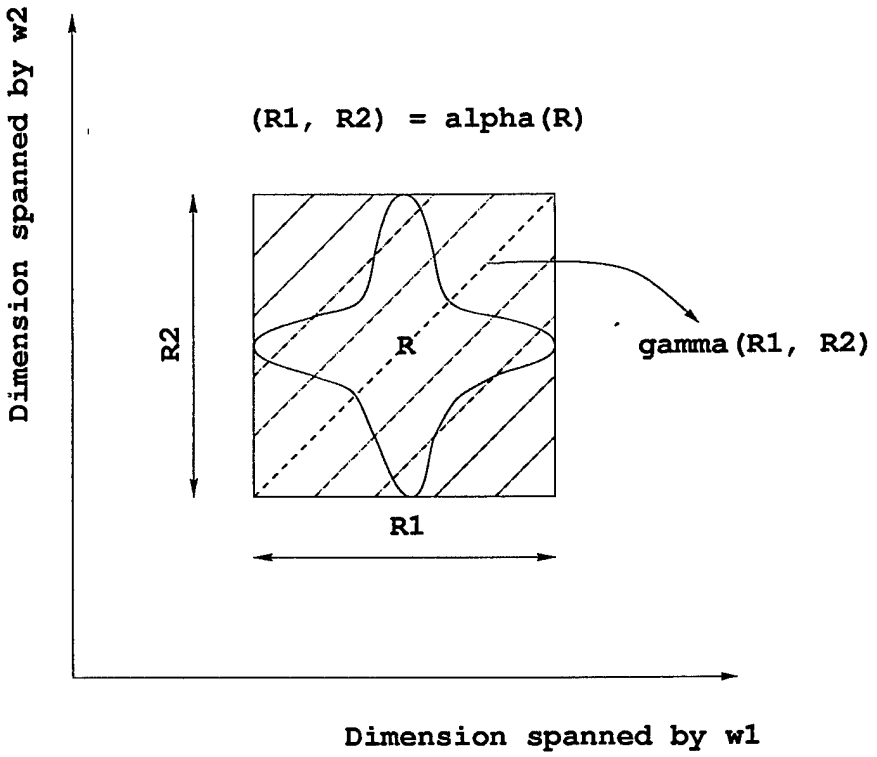


Figure 3.1: Concretization of projection of R is a superset of R

Now $\alpha_1(R) = \exists(\mathbf{x} - w_1).R(\mathbf{x})$ *which reduces to* $\exists x_4 \cdot R(\mathbf{x})$. *Existential hiding of variable* x_4 *from* $R(\mathbf{x})$ *results in* $\alpha_1(R) = \bar{x_1}\bar{x_2}\bar{x_3} \vee x_1\bar{x_2}\bar{x_3} \vee \bar{x_1}x_2\bar{x_3} \vee \bar{x_1}\bar{x_2}x_3$. *This is the* atmost_one *function among the variables* $\{x_1, x_2, x_3\}$, *which is true if and only if at most one of the variables in* $\{x_1, x_2, x_3\}$ *is true. Similarly* $\alpha_2(R) = \bar{x_2}\bar{x_3}\bar{x_4} \vee x_2\bar{x_3}\bar{x_4} \vee \bar{x_2}x_3\bar{x_4} \vee \bar{x_2}\bar{x_3}x_4$.

Lemma 1 *For every predicate* $R(\mathbf{x})$ *and collection of subsets* $(w_1, \ldots, w_p)$ *of* $\mathbf{x}$, $R \rightarrow \gamma(\alpha(R))$.

Proof: *The proof for Lemma 1 is simple since for each* j, $1 \leq j \leq p, (R \rightarrow \alpha_j(R))$. *Further, it is a simple fact of propositional logic that* $\wedge_j(p \rightarrow q_j)$ *implies that* $p \rightarrow \wedge_j q_j$.

The intuition behind the proof is that each individual projection $\alpha_j(R)$ is an over-approximation of R. Furthermore, the conjunction of a set of over-approximations of R is also an over-approximation of R.

Lemma 1 effectively states that projecting a predicate R onto a collection of subsets and then concretizing the projections by γ results in an *over-approximation.* Figure 3.1 gives a geometric interpretation of this observation.

Example 3 *Continuing on example 2, concretizing the projections of the* one_hot *function through* γ *results in* $\gamma(\alpha(R)) = \bar{x_1}\bar{x_2}\bar{x_3}\bar{x_4} \vee x_1\bar{x_2}\bar{x_3}\bar{x_4} \vee \bar{x_1}x_2\bar{x_3}\bar{x_4} \vee \bar{x_1}\bar{x_2}x_3\bar{x_4} \vee \bar{x_1}\bar{x_2}\bar{x_3}x_4 \vee x_1\bar{x_2}\bar{x_3}x_4$. *Note that apart from the minterms that are in the onset of the original* one_hot *function* $R(\mathbf{x})$, *this also includes the minterms* $\bar{x_1}\bar{x_2}\bar{x_3}\bar{x_4}$ *and* $x_1\bar{x_2}\bar{x_3}x_4$. *Thus, the onset of the resulting function after projection and concretization is a superset of what we started with.*

Meet and Join: Let $\mathbf{R} = (R_1, \ldots, R_p)$ and $\mathbf{S} = (S_1, \ldots, S_p)$ be two equally sized tuples. We define the *meet* ($\sqcap$) and *join* ($\sqcup$) operator between $\mathbf{R}$ and $\mathbf{S}$ as follows:

$$\begin{aligned}(R_1, \ldots, R_p) \sqcap (S_1, \ldots, S_p) &= (R_1 \wedge S_1, \ldots, R_p \wedge S_p) \\ (R_1, \ldots, R_p) \sqcup (S_1, \ldots, S_p) &= (R_1 \vee S_1, \ldots, R_p \vee S_p)\end{aligned}$$

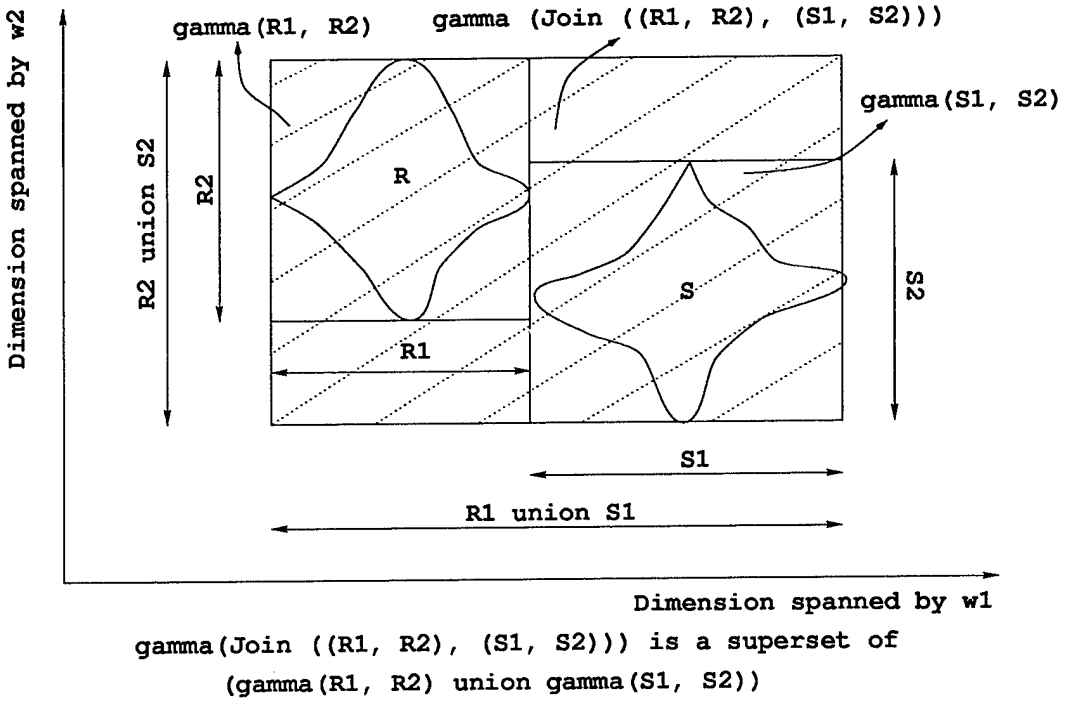


Figure 3.2: Geometric interpretation of join operator ($\sqcup$)

Note that $\gamma(\mathbf{R}) \cup \gamma(\mathbf{S}) \subseteq \gamma(\mathbf{R} \sqcup \mathbf{S})$, which makes the join operator an approximation of set union. Figure 3.2 gives a geometric interpretation of the *join* operator, illustrating the approximation induced by it. The meet operator, however, is an exact set intersection operator, since $\gamma(\mathbf{R}) \cap \gamma(\mathbf{S}) = \gamma(\mathbf{R} \sqcap \mathbf{S})$.

The operator α allows us to represent a big BDD with support in $\mathbf{x}$ by a tuple of potentially smaller BDDs with limited support, at the cost of a loss of accuracy. Concretization through γ can potentially result in a bigger BDD with bigger support, hence we would like to avoid computing $\gamma(R_1, \ldots, R_p)$ explicitly. Let Im_{ap} (the subscript *ap* denotes "approximate") return the projected version of the image of an *implicit* conjunction of BDDs, and let Pre_{ap} return the projected version of the preimage of an *implicit* conjunction of BDDs. Figure 3.3 gives a geometric interpretation of Im_{ap}, and Figure 3.4 gives a geometric interpretation of Pre_{ap}.

$$Im_{ap}(\mathbf{R}, \mathbf{n}) = \alpha(Im(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y})))$$
$$Pre_{ap}(\mathbf{R}, \mathbf{n}) = \alpha(Pre(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y})))$$

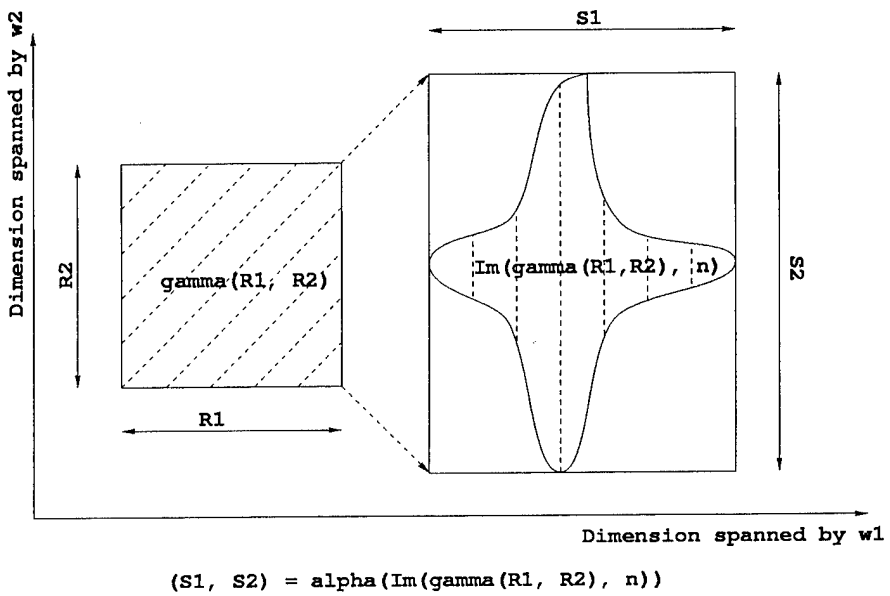


Figure 3.3: Geometric interpretation of $Im_{ap} : (S_1, S_2) = Im_{ap}((R_1, R_2), \mathbf{n})$

Using Im_{ap}, we can compute an over-approximation, of the reachable states in a machine M, through a least fix-point iteration.

$$FwdReach_{ap}(q_0) = \text{ lfp } \mathbf{R}.(\alpha(q_0) \sqcup Im_{ap}(\mathbf{R}, \mathbf{n}))$$

The notation *lfp* is short for the following piece of pseudo-code.

$\mathbf{R_{reached}} \leftarrow \alpha(q_0)$
$\mathbf{R_{previous}} \leftarrow \alpha(0)$
while $\mathbf{R_{reached}} \neq \mathbf{R_{previous}}$ **do**
  $\mathbf{R_{previous}} \leftarrow \mathbf{R_{reached}}$
  $\mathbf{R_{reached}} \leftarrow \alpha(q_0) \sqcup Im_{ap}(\mathbf{R_{previous}}, \mathbf{n})$
return $\mathbf{R_{reached}}$

Thus, the least fixed point iteration [7] starts with $\mathbf{R} = (0, \ldots, 0)$, and on each iteration *joins* the current approximate set with the approximate successor set. Finally, after reaching convergence, it returns a tuple $\mathbf{R}$ to $FwdReach_{ap}(q_0)$. A superset of the set of states that can be reached from the initial states is the *implicit* conjunction: $\gamma(FwdReach_{ap}(q_0))$.

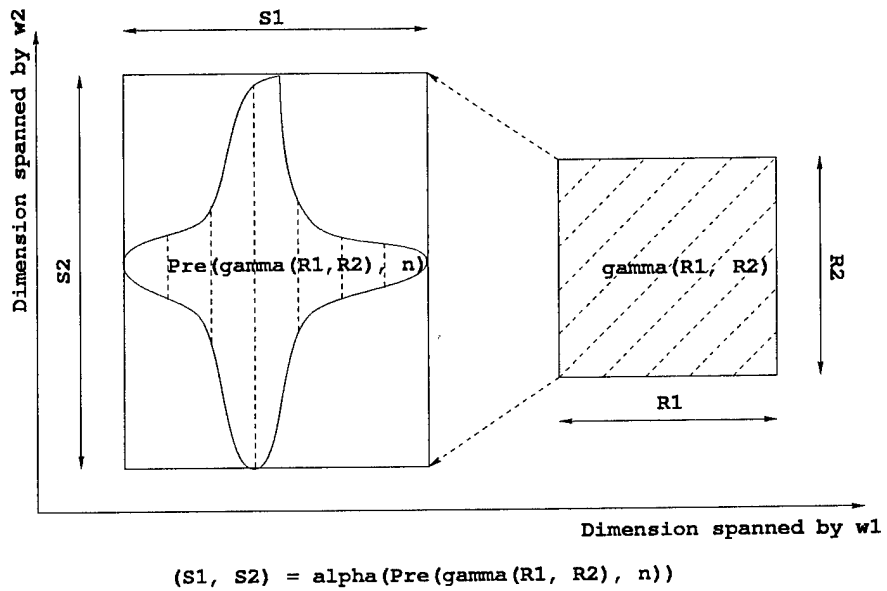


Figure 3.4: Geometric interpretation of $Pre_{ap} : (S_1, S_2) = Pre_{ap}((R_1, R_2), \mathbf{n})$

Similarly, let g be the set of states that satisfy a user-provided property, and let $g(\mathbf{x})$ be the BDD representing it. The BDD $\neg g(\mathbf{x})$ represents the set of states violating the property. Using Pre_{ap}, we can compute an over-approximation of the set of states in M that can reach some state in the set $\neg g$, as follows:

$$BackReach_{ap}(\neg g) = \text{ lfp } \mathbf{R}.(\alpha(\neg g) \sqcup Pre_{ap}(\mathbf{R}, \mathbf{n}))$$

The notation *lfp* is short for the following piece of pseudo-code.

$\mathbf{R_{reached}} \leftarrow \alpha(\neg g)$
$\mathbf{R_{previous}} \leftarrow \alpha(0)$
while $\mathbf{R_{reached}} \neq \mathbf{R_{previous}}$ **do**
  $\mathbf{R_{previous}} \leftarrow \mathbf{R_{reached}}$
  $\mathbf{R_{reached}} \leftarrow \alpha(\neg g) \sqcup Pre_{ap}(\mathbf{R_{previous}}, \mathbf{n})$
return $\mathbf{R_{reached}}$

A superset of the set of states that can reach $\neg g$ is is the *implicit* conjunction: $\gamma(BackReach_{ap}(\neg g))$.

Using Lemma 1 and monotonicity of Im and Pre functions, it can be shown that

the derived functions Im_{ap} and Pre_{ap} have the property

$$Im(R(\mathbf{x}), \mathbf{n}) \subseteq Im(\gamma(\alpha(R(\mathbf{x}))), \mathbf{n}) \subseteq \gamma(Im_{ap}(\alpha(R(\mathbf{x})), \mathbf{n}))$$
$$Pre(R(\mathbf{x}), \mathbf{n}) \subseteq Pre(\gamma(\alpha(R(\mathbf{x}))), \mathbf{n}) \subseteq \gamma(Pre_{ap}(\alpha(R(\mathbf{x})), \mathbf{n}))$$

Theorem 1 *For a given Mealy machine M,*

$$FwdReach(q_0) \rightarrow \gamma(FwdReach_{ap}(q_0))$$
$$BackReach(\neg g) \rightarrow \gamma(BackReach_{ap}(\neg g))$$

Proof: *Proof for Theorem 1 follows trivially from the previous two equations, as every iteration of the approximate least fix-point routine for $FwdReach_{ap}(q_0)$ and $BackReach_{ap}(\bar{g})$ is an over-approximation.*

Today's designs have a very large set of state variables, $\mathbf{x} = \{x_1, \ldots, x_k\}$. There are 2^{2^k} possible Boolean functions over these variables. In the worst case, the BDD size for some function over these variables is $O(2^k)$. For large k, this worst case size may be prohibitively expensive. However in our scheme of approximation with the choice of subsets $\mathbf{w} = (w_1, \ldots, w_p)$, the support sets of the intermediate BDDs while computing $FwdReach_{ap}(q_0)$ and $BackReach_{ap}(\neg g)$ are restricted within w_j, where $w_j \subseteq \mathbf{x}$. In particular, if $|w_j| = m$ (we use $|w_j|$ to denote the cardinality of the set w_j), and $m < k$, then the worst case BDD size of $O(2^m)$ is more amenable than the worst case size of $O(2^k)$. These smaller sized support set BDDs result in robust and scalable verification algorithms. However, the price paid for robustness is that any interaction or correlation between state variables in different subsets is lost in the process.

Definition 1 *A collection of subsets $\mathbf{w}'$ is a refinement of the collection $\mathbf{w}$ if each block of $\mathbf{w}$ can be expressed as a union of blocks of $\mathbf{w}'$.*

Lemma 2 *If $\mathbf{w}'$ is a refinement of $\mathbf{w}$ and let $\langle \alpha', \gamma' \rangle$ and $\langle \alpha, \gamma \rangle$ be associated with $\mathbf{w}'$ and $\mathbf{w}$ respectively, then*

$$\gamma(\alpha(R)) \rightarrow \gamma'(\alpha'(R))$$

From Lemma 2 and monotonicity of the predicate transformer Im_{ap}, we conclude that *coarser* collection of subsets gives tighter approximations. At the same time, coarser collection of subsets entails that the intermediate image BDDs would have larger support sets, making them more liable to BDD blowup problems. (As expected in the limit case, when there is just one subset, $w_1 = \mathbf{x}$, in the collection $\mathbf{w}$, the algorithms $FwdReach_{ap}$ and $BackReach_{ap}$ give exact results.)

Earlier schemes of approximation required that the various subsets in the collection $\mathbf{w}$ be mutually disjoint partitions. In the next section we show that with overlapping projections we can obtain tighter approximations with smaller sized subsets than with disjoint partitions.

3.2.2 Why Overlapping Projections?

Overlapping projections can capture limited interactions among state machines while keeping the sizes of the BDDs under control. We discuss some common scenarios where this happens in this section. In contrast, disjoint partitions can only capture interactions among a set of state machines by including all of them in a single projection, which often leads to large variable subsets that cause BDD size blowup.

Typical designs today exhibit the following phenomena:

- *Small Interface Phenomena*: Often, two rather big state machines have a *small interface*. Figure 3.5 is one way to visualize the phenomena. The next state transitions of the big machine M_1 may depend on only a few of the state bits of M_2 and similarly the next state transitions of the big machine M_2 may depend on only a few of the state bits of M_1. This interaction between these big machines can be captured by choosing one subset which includes all of the state bits of M_1 and a few of the state bits of M_2 relevant to M_1. Another subset would have all of the state bits of M_2 and a few of the states bits of M_1 relevant to M_2. Capturing the same interaction with disjoint subsets would require including all of the state variables of the big machines M_1 and M_2 in a single subset. Such large subsets are very susceptible to BDD blowup problems.

- *Master-Slave Phenomena*: Design modules usually have a *master* FSM that

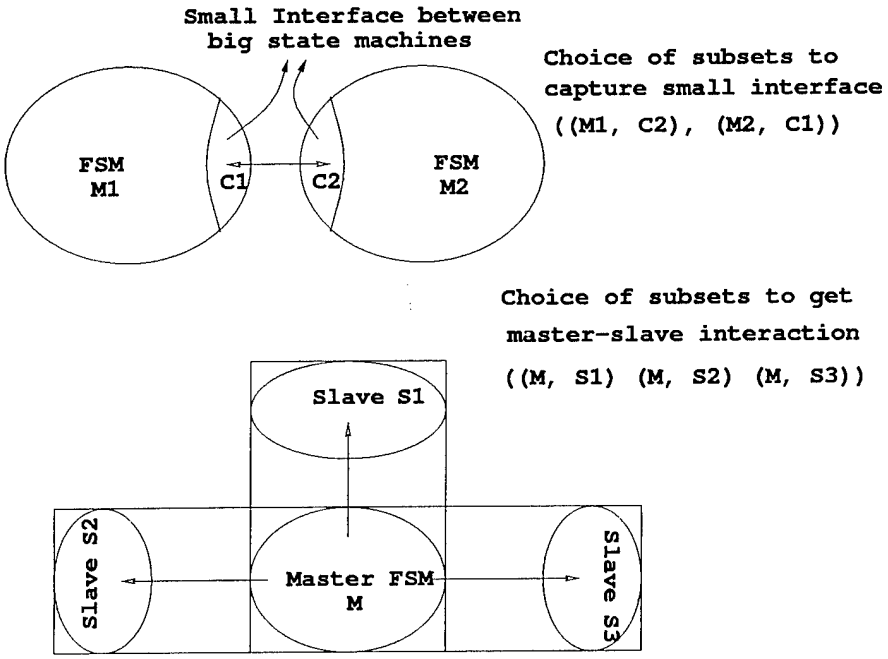


Figure 3.5: Capturing interaction b/w FSMs with overlapping projections

communicates with a number of other *slave* FSMs. This interaction between the master and each of its individual slaves can be captured by having subsets where the master is paired with each of its slaves in *different* blocks. Figure 3.5 is one way to visualize the phenomena. Once again, capturing this same interaction with disjoint subsets would require including the state variables of the large master FSM and the state variables of *all* the slave machines in a single subset. Such large subsets are very susceptible to BDD blowup problems.

3.2.3 Projections *vs* Partitions

Clearly the scheme of approximation by projections is more general than that which uses partitions (since disjoint partitions can be viewed as a special case of projections, where there is no overlap). As seen in the earlier section, disjoint partitions require larger sized blocks to capture the same property. Thus, overlapping subsets allow us to hit intermediate points in the *Quality of Result* vs *memory space* tradeoff curve (Figure 3.6), with disjoint partitions on one extreme and exact reachability on the

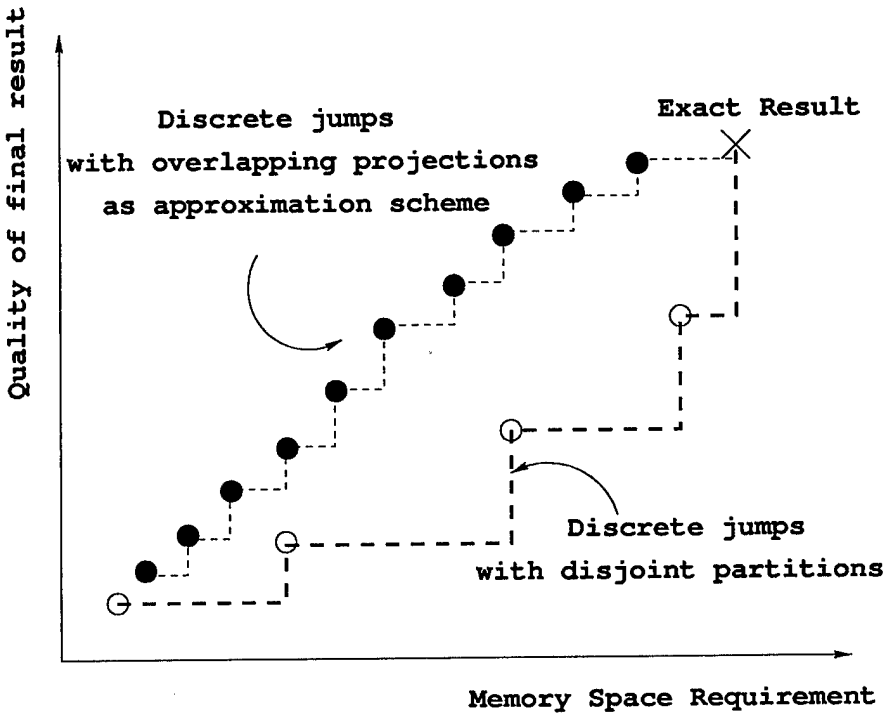


Figure 3.6: Quality of result *vs* memory requirement tradeoff curve

other. (Since the improvement in the quality of the results happens in discrete steps the curves for disjoint and overlapping schemes in Figure 3.6 are shown with dotted lines as step functions.)

With disjoint partitions as the underlying approximation scheme, the sizes of the individual subsets have to increase substantially (which entails a substantial increase in memory space requirements) before the quality of the approximation can improve. On the other hand, with overlapping projections as the underlying approximation scheme, we can incrementally increase the size of the subsets and incrementally improve the quality of result. In particular, note in Figure 3.6, the curve for overlapping projections is consistently above that for disjoint subsets. Therefore, we expect to get better quality results at a lower memory space requirement cost.

3.3 Related Work

At a high level, this idea of approximate forward and backward traversals is quite similar to that of Wong-Toi *et al.* [22], who used successive forward and backwards over-approximations and under-approximations to verify real-time systems. That work used polyhedra for representing sets of real numbers along with BDDs, but approximation was used only for the polyhedra, not for the BDDs.

Various approaches to approximate reachability and verification using BDDs have preceded this work. Ravi *et al.* [59, 60] used "high density" BDDs to compute an *under-approximation* of the forward reachable set.

Cho *et al.* [12] proposed symbolic forward reachability algorithms that induce an *over-approximation.* Their basic idea was to partition the set of state bits into *mutually disjoint* subsets, and then to do symbolic forward propagation on the individual subsets. The individual subsets can be viewed as sub-machines which have in some ways been torn from other sub-machines. The original problem is thus reduced to doing exact symbolic forward propagation over smaller sub-machines. This induces extra degrees of freedom for the sub-machines, and hence yields an over-approximation of the reachable state space. They also proposed different variants of the approximated symbolic forward propagation algorithm: MBM (Machine By Machine) and FBF (Frame By Frame) which basically differ in the way they model the interaction among the various sub-machines. FBF allows interactions among the sub-machines at each time frame of a least fixed point routine, and hence allows for tighter don't care *sequences* to constrain the other sub-machines. MBM on the other hand allows interaction only after a complete least fixed point has been computed for a sub-machine. As a result, the sequencing information is lost when trying to constrain the other sub-machines. They further proposed two variants of the FBF scheme, RFBF (Reached Frame By Frame) and TFBF (To Frame By Frame), which again differ in the constraint set posed to the various sub machines during the course of the least fixed point routine. Cho *et al.* [11] also proposed heuristics on how to partition the set of state bits.

Moon *et al.* [53] used approximate reachability algorithms from [12] to aid model

checking algorithms, and Cabodi *et al.* [8] combined approximate forward reachability with *exact* backward reachability. Lee *et al.* [47] proposed "tearing" schemes to do approximate symbolic backward reachability. They proposed "variable tearing" and "block tearing" schemes to approximate the next state relation of a system, and then incrementally refine the next state relation until it is sufficient to prove/disprove a given ACTL or ECTL [51] property. They also partitioned the set of state bits into *mutually disjoint* subsets. They formed the block sub-relations for the various subsets, and then incrementally "stitched" the block sub-relations together until the approximated next state relation was accurate enough to prove or disprove a given property.

In contrast to the approaches in [59, 60] we are interested in computing over-approximations (supersets). In contrast to the approaches in [8, 12, 11, 47, 53], we allow for overlapping subsets of the variables. Our research [29, 30, 31, 32, 34] shows that overlapping projections are a more refined approximation scheme compared to earlier schemes based on disjoint partitions.

3.4 Conclusions

This chapter introduced and defined the idea of overlapping projections as an approximation scheme. Earlier schemes of approximation based on disjoint subsets can be viewed as a special case. Overlapping projections is a more robust scheme since it allows for capturing interaction between different state machines with smaller memory space requirements.

3.5 Appendix: Galois Connections

Galois connection is a general framework to define many approximate methods. In this section, we briefly explain the Galois connections framework, and show how approximation with overlapping projections can be explained under this framework.

Approximation methods allow us to prove properties in the simpler mathematical model and conclude that the more concrete model has some related properties.

Property preservation between the two models can be formally justified by showing a formal relation between the two models, using the theory of abstract interpretation.

Definition 2 : *Given a class of temporal properties $\mathcal{P}$ and two systems $\mathcal{S}$ and $\mathcal{S}_A$ (where $\mathcal{S}_A$ is an abstract (simpler) version of $\mathcal{S}$), we say that:*

- *$\mathcal{S}_A$ is a weakly preserving abstraction of $\mathcal{S}$ for $\mathcal{P}$ if for each $\phi_A \in \mathcal{P}$ (where ϕ_A is an abstract version of ϕ)*

$$\text{if } \mathcal{S}_A \models \phi_A \text{ then } \mathcal{S} \models \phi$$

- *$\mathcal{S}_A$ is strongly preserving abstraction of $\mathcal{S}$ for $\mathcal{P}$ if for any $\phi_A \in \mathcal{P}$,*

$$\mathcal{S}_A \models \phi_A \text{ if and only if } \mathcal{S} \models \phi$$

Strong preservation does not leave much room for generating simpler systems. For example, if all CTL [15] properties have to be preserved, the two systems $\mathcal{S}$ and $\mathcal{S}_A$ must be bisimilar. Thus, weak preservation is more often used. Not only does weak preservation allow us to deal with relatively simpler systems, it also allows preservation of properties over the $\forall$CTL fragment [49] of CTL logic.

Using the terminology of [19], an abstract system is defined in terms of an abstract domain, which is a set of states Σ_A that includes a partial order $\preceq$, where $a_1 \preceq a_2$ if a_1 is a "more precise" abstract state than a_2. Such abstractions are frequently presented in terms of Galois connections. Here abstract states represent sets of concrete states, and the two posets are the power sets of concrete states $\mathcal{P}(\Sigma)$, ordered by set inclusion, and an abstract domain Σ_A ordered by $\preceq$.

Definition 3 : *Let $(\Sigma_A, \preceq)$ and $(\Sigma, \subseteq)$ be two partially ordered sets. A Galois connection between these posets is a pair of functions $\langle \alpha, \gamma \rangle$, where the abstraction function $\alpha : \mathcal{P}(\Sigma) \rightarrow \Sigma_A$ and the concretization function $\gamma : \Sigma_A \rightarrow \mathcal{P}(\Sigma)$ satisfy the following properties :*

1. *α and γ are monotone*

$$\begin{aligned} S_1 \subseteq S_2 &\rightarrow \alpha(S_1) \preceq \alpha(S_2) \\ a_1 \preceq a_2 &\rightarrow \gamma(a_1) \subseteq \gamma(a_2) \end{aligned}$$

2. *result of abstraction followed by concretization is something larger*

$$\begin{aligned} \forall S \in \mathcal{P}(\Sigma).\ S \subseteq \gamma(\alpha(S)) \\ \forall a \in \Sigma_A.\ a \preceq \alpha(\gamma(a)) \end{aligned}$$

These two conditions can be alternatively written as $\alpha(a) \preceq b$ iff $a \subseteq \gamma(b)$. In the special case where $\forall a \in \Sigma_A, \alpha(\gamma(a)) = a$, the pair $\langle \alpha, \gamma \rangle$ is called a *Galois insertion.*

3.5.1 Typical Applications of Galois Connections

Galois connections can be used in two different ways:

- *Proving Properties*: Given a system $\mathcal{S}$ and temporal property ϕ, choose appropriate poset $(\mathcal{S}_A, \preceq)$, a Galois connection $\langle \alpha, \gamma \rangle$, and the weakest ϕ_A (with respect to $\preceq$) such that

$$\begin{aligned} \mathcal{S}_A &\models \phi_A \\ \gamma(\phi_A) &\subseteq \phi \end{aligned}$$

  The proof that these two conditions are sufficient to conclude $\mathcal{S} \models \phi$ is left to the interested reader.

- *Generate Invariants*: Given a system $\mathcal{S}$, choose an appropriate poset $(\mathcal{S}_A, \preceq)$, a Galois connection $\langle \alpha, \gamma \rangle$ and generate ϕ_A by doing fix-point computation in $\mathcal{S}_A$. The assertion $\gamma(\phi_A)$ is an invariant of the concrete system $\mathcal{S}$.

3.5.2 Overlapping Projections as a Galois Connection

In our applications, the pair of functions $\langle \alpha, \gamma \rangle$ defined in Section 3.1 form a Galois connection. The partially ordered set describing the concrete space is $([\mathbf{x} \rightarrow \mathcal{B}], \subseteq)$, and the poset describing the abstract space is $(\mathcal{P}([w_1 \rightarrow \mathcal{B}]) \times \ldots \times \mathcal{P}([w_p \rightarrow \mathcal{B}]), \sqsubseteq)$. Note that $\mathcal{P}(S)$ denotes the power set of S, and the ordering relation for the abstract space is defined as $(R_1, \ldots, R_p) \sqsubseteq (S_1, \ldots, S_p)$ *iff* $\forall i \in [1 \ldots p]\ (R_i \rightarrow S_i)$.

Given the ordering relation ($\sqsubseteq$) in the abstract domain, it is easy to verify that the *join* operator returns the *least upper bound*, and *meet* returns the *greatest lower bound* of the two elements $\mathbf{R}$ and $\mathbf{S}$ in the abstract domain. In our applications, since $\forall a \in \Sigma_A, \alpha(\gamma(a)) = a$, the pair $\langle \alpha, \gamma \rangle$ is actually a *Galois insertion*. Furthermore, as illustrated in Section 3.5.1, we use this scheme of approximation to prove safety properties and to automatically generate invariants.

Chapter 4

Approximate Forward Reachability

This chapter first defines the key challenge in symbolic forward reachability, namely, the image computation problem. Existing methods of BDD-based image computation are briefly reviewed. A new image computation method, called *multiple constrain*, is defined. This method allows for efficient computation of projections of exact images. Finally, the results obtained by applying this method to different design examples are presented.

4.1 Basic Algorithm

Computing the set of states that can be reached from the initial states is at the heart of model checking. In Section 2.3, we saw how BDDs can be used to represent Boolean functions, sets of states, and relations. This enables modeling synchronous hardware designs with BDDs. Let q_0 be the set of initial states, represented by a BDD $q_0(\mathbf{x})$. We wish to compute a BDD, $R(\mathbf{x})$, that represents the states reachable from q_0 via the transition relation T (represented by the BDD $T(\mathbf{x}, \mathbf{x}')$). We first consider the problem of finding those states R_1, reachable in at most one step from q_0. This set

of states is given by

$$R_1 = q_0 \cup \{s' \mid \exists s \cdot [s \in q_0 \wedge (s, s') \in T]\}$$

Given the BDDs $q_0(\mathbf{x})$ and $T(\mathbf{x}, \mathbf{x}')$, we can compute a BDD representing R_1 by performing the logical operations corresponding to the above expression:

$$R_1(\mathbf{x}') = q_0(\mathbf{x}') \vee \exists \mathbf{x} \cdot [q_0(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}')].$$

Similarly, the set of states reachable in at most two steps is represented by

$$R_2(\mathbf{x}') = q_0(\mathbf{x}') \vee \exists \mathbf{x} \cdot [R_1(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}')].$$

In general, the set of states reachable in at most $n+1$ steps is represented by

$$R_{n+1}(\mathbf{x}') = q_0(\mathbf{x}') \vee \exists \mathbf{x} \cdot [R_n(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}')].$$

Note that each set of states is a superset of the previous one. Since the total number of states in a hardware design is finite, at some point we must have $R_{n+1} = R_n$. No further states are reachable, so the set of all reachable states is represented by $R_n(\mathbf{x})$.

4.2 Methods to Compute Images

The key step in the high level algorithm outlined earlier is computing the one step successors of a set of states. This is widely referred to as *image* computation. Existing methods of BDD-based image computation can be broadly classified into two categories: transition relation approach and transition function approach.

4.2.1 Transition Relation Approach

As outlined in the previous section, this method relies on building BDDs to represent the transition relation $T(\mathbf{x}, \mathbf{x}')$ of the circuit. The key problem is computation of the

relational product:

$$R'(\mathbf{x}') \;=\; \exists \mathbf{x} \cdot [R(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{x}')].$$

Although the relational product can be computed using the normal BDD algorithms for restriction and Boolean connectives, it does not work well in practice for large designs. This is because the basic algorithm requires having $T(\mathbf{x}, \mathbf{x}')$ be a monolithic relation, consisting of a single BDD. Unfortunately, for most practical designs, this BDD is very large. It is much more efficient to use a special purpose algorithm, based on partitioned transition relations [7].

Partitioned Transition Relations

Recall that for synchronous circuits the transition relation $T(\mathbf{x}, \mathbf{x}')$ is basically the conjunction of a number of relations $t_i(\mathbf{x}, x_i')$. The individual t_i relations have small BDD representations, because they describe the constraints that a *single* next state variable x_i' must satisfy in a legal transition. (On the other hand, the monolithic BDD for the transition relation $T(\mathbf{x}, \mathbf{x}')$ describes the constraints that *all* next state variables must satisfy in a legal transition, and is invariably extremely big for practical designs.) Instead of forming the conjunction of the $t_i(\mathbf{x}, x_i')$'s, we can represent the circuit by a list of these BDDs, which are *implicitly* conjuncted. Such a list is called *partitioned transition relation* [7]. The relational product computation problem is now of the form

$$R'(\mathbf{x}') \;=\; \exists \mathbf{x} \cdot [R(\mathbf{x}) \wedge (t_1(\mathbf{x}, x_1') \wedge t_2(\mathbf{x}, x_2') \wedge \ldots \wedge t_n(\mathbf{x}, x_n'))].$$

The main difficulty in computing $R'(\mathbf{x}')$ without building the conjunction is that existential quantification does not distribute over conjunction.

With such partitioned transition relations, a technique called *early quantification*, is used to allow for more efficient relational product computations. The early quantification technique is based on two observations. First, circuits exhibit locality, so many of the $t_i(\mathbf{x}, x_i')$ will depend on only a small number of the variables in $\mathbf{x}$. Second,

although existential quantification does not distribute over conjunction, sub-formulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. Thus, we can conjoin the $t_i(\mathbf{x}, x_i')$ with $R(\mathbf{x})$ one at a time and use *early quantification* to quantify out those state variables from $\mathbf{x}$ when none of the remaining $t_j(\mathbf{x}, x_j')$ depend on those state variables.

The impact of this method clearly depends on the order in which the various t_i relations are conjuncted with $R(\mathbf{x})$. The hope is that with many of the state variables being quantified early, the intermediate BDDs will have smaller and more manageable support sets. More formally, we could define the *lifetime* of a state variable as the the interval $[i, j]$ where i is the least index of relation t_i where it appears, and j is the greatest index of relation t_j where it appears. The goal then is to sort the various individual t_i relations so that the maximum number of live variables at any point is minimized. Based on this idea, different heuristics to order the various $t_i(\mathbf{x}, x_i')$ in the list forming the partitioned transition relation have been proposed [28].

While a partitioned transition relation with one BDD for each state variable is almost always more efficient than constructing a monolithic transition relation, it may not be the best choice. As long as the BDDs do not become too large, it is better to combine some of the $t_i(\mathbf{x}, x_i')$ into one BDD by forming their conjunction. Fewer BDD nodes may be needed in this representation if the BDDs for the individual t_i's that are combined have a similar structure near the root of their BDDs. Combining some of the BDDs in a partitioned transition can also speed up the algorithms for model checking and reachability analysis. Ranjan *et al.* [61] proposed heuristics on how to combine some of the individual t_i relations into *clusters*. The next state transition relation is then represented as a list of BDDs where each BDD in the list is the transition relation for a cluster of state variables. The BDDs for these clusters are further ordered in the list by a heuristic to allow for early quantification benefits. More details can be obtained from [61]. This transition relation approach did not work well on our larger examples.

4.2.2 Transition Function Approach

Instead of using transition *relations* for image computation, Coudert and Madre [18] proposed a way of computing the image of a set by merely manipulating the vector of next state functions $\mathbf{n}$. (Following the notation of Section 2.4, we use $Im(R(\mathbf{x}), \mathbf{n})$ to denote the BDD for image of a vector of Boolean functions $\mathbf{n} : [n_1, \ldots, n_k]$, when the domain is restricted to the set represented by the BDD $R(\mathbf{x})$.) Coudert and Madre proposed two algorithms to compute the BDD for $Im(R(\mathbf{x}), \mathbf{n})$ without computing the transition relation.

Both the algorithms are based on the *constrain operator*, which was defined in Section 2.5. The fundamental property of constrain can be expressed by the following theorem:

Theorem 2 *Let* $\mathbf{f} = [f_1, \ldots, f_n]$ *be a vector of functions, where the individual functions* f_i *are represented by a BDD* $f_i(\mathbf{x})$. *Let* R *be a non-empty set represented by the BDD* $R(\mathbf{x})$. *Let* $\mathbf{f} \downarrow R(\mathbf{x})$ *be a new vector of functions defined as*

$$\mathbf{f} \downarrow R(\mathbf{x}) \quad =_{def} \quad [f_1 \downarrow R, \ldots, f_n \downarrow R].$$

Then image of R *under* $\mathbf{f}$ *is equal to the range of the vector of functions,* $\mathbf{f} \downarrow R$, i.e.

$$Im(R(\mathbf{x}), \mathbf{f}) = Im(1, \mathbf{f} \downarrow R).$$

Proof: *Details of the proof for this theorem can be obtained from [18].*
Both the algorithms proposed by Coudert and Madre [18] work in two steps:

1. first step is common to both algorithms and involves computing a new vector of functions $\mathbf{n}' : [n'_1, \ldots, n'_k]$ such that $Im(R(\mathbf{x}), \mathbf{n}) = Im(1, \mathbf{n}')$. From theorem 2, it is easy to see that the vector $\mathbf{n}'$ can be computed as $\mathbf{n} \downarrow R$.

2. the second step consists of computing a BDD for $Im(1, \mathbf{n}')$ by using *co-domain partitioning* in the first algorithm and *domain partitioning* in the second.

Codomain Partitioning Algorithm

This algorithm uses the constrain operator to partition the co-domain of the vectorial function $\mathbf{n}'$ to compute $Im(1, \mathbf{n}')$. The algorithm is a direct application of the following theorem.

Theorem 3 *Let* $\mathbf{x} = \{x_1, \ldots, x_n\}$ *be the set of state variables, where each variable* x_i *has an associated next state function* f_i. *Let* $\mathbf{f_n} = [f_1, \ldots, f_n]$ *be a vector of functions, where the individual function* f_i *is represented by a BDD* $f_i(\mathbf{x})$. *Then*

$$Im(1, \mathbf{f_n}) \;=\; Im(1, \mathbf{f_{n-1}} \downarrow f_n) \wedge x_n \;\vee\; Im(1, \mathbf{f_{n-1}} \downarrow \neg f_n) \wedge \neg x_n.$$

Proof: *Details of the proof for this theorem can be obtained from [18].*
Since we will be using the codomain partitioning algorithm in later parts of this thesis, we describe it below in more detail. Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be the set of state variables, where each variable x_i has an associated next state function f_i. The vector of functions, $\mathbf{f_n} = [f_1, \ldots, f_n]$, whose range is desired, is passed as an argument.

function *Codomain_Split*($\mathbf{f_n} = [f_1, \ldots, f_n]$)
begin
  if ($\mathbf{f_n} = [f_1]$)
    if ($f_1 = 1$) **return** x_1
    elsif ($f_1 = 0$) **return** $\bar{x_1}$
    else return 1
  else let $\mathbf{f_n} = [f_1, \mathbf{f_{n-1}}]$
    if ($f_1 = 1$) **return** $x_1 \wedge$ *Codomain_Split*($\mathbf{f_{n-1}} \downarrow f_1$)
    elsif ($f_1 = 0$) **return** $\neg x_1 \wedge$ *Codomain_Split*($\mathbf{f_{n-1}} \downarrow \neg f_1$)
    else return (*ite* x_1 *Codomain_Split*($\mathbf{f_{n-1}} \downarrow f_1$) *Codomain_Split*($\mathbf{f_{n-1}} \downarrow \neg f_1$))
end

The number of recursions needed to compute $Im(1, \mathbf{f_n})$ is bound by the number of elements of the vector $\mathbf{f_n}$. Several techniques have been proposed [18] to reduce the number of recursions through dynamic programming.

Domain Partitioning Algorithm

This algorithm uses the constrain operator to partition the domain of the vectorial function $\mathbf{n}'$ to compute $Im(1, \mathbf{n}')$. The algorithm is a direct application of the following theorem.

Theorem 4 *Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be the set of state variables, where each variable x_i has an associated next state function f_i. Let $\mathbf{f_n} = [f_1, \ldots, f_n]$ be a vector of functions, where the individual function f_i is represented by a BDD $f_i(\mathbf{x})$. Then*

$$Im(1, \mathbf{f_n}) \;=\; Im(1, \mathbf{f_{n-1}} \downarrow x_n) \;\vee\; Im(1, \mathbf{f_{n-1}} \downarrow \neg x_n).$$

Proof: *Details of the proof for this theorem can be obtained from [18].*

Discussion and Comparison

Each of the methods for image computation have their strengths and their weaknesses. There are some circuits for which transition relation based methods seem to work best, while there are some circuits for which the transition function based methods work better. Circuits which have distributed independent blocks that have small local influence are amenable to early quantification benefits. Hence, transition relation based schemes would work better on them. But if a circuit has almost every state variable depending on almost every other state variable, then early quantification cannot help and the intermediate BDDs can get prohibitively large. Under these scenarios, sometimes a smart choice of splitting variables can help the transition function perform a lot better. The following example brings out this point.

Example 4 [1] *Consider a* barrel-shifter *with select lines $sel_1, \ldots, sel_n$ and a data register $a_1, \ldots, a_m$ (where $m = 2^n$). The select line bits decide by how many positions the data in the register should be shifted. Clearly, the next state function for each state bit in the data register now depends on every other bit of the register. This means that early quantification cannot help. However if we use the transition function method,*

[1] We thank Ken McMillan for suggesting this simple example.

and first split on the $sel_1, \ldots, sel_n$ variables, immediately all the next state functions reduce to very simple functions, whose image can be easily computed.

- As a rough rule of thumb, transition relation methods usually have bigger intermediate BDDs compared to the transition function method.
- However, the transition function methods are often slow. This is because *memoization* is often ineffective, because the probability of seeing the exact same vector of functions is low.
- However, in our application domain of approximate reachability, the number of codomain variables is much smaller than the number of domain variables. In this scenario, we found that the codomain partitioning algorithm worked better than the transition relation method, on the design examples used in this thesis.
- Unlike the transition relation method which requires maintaining one copy of the state variables to denote the present state $\mathbf{x}$, and another copy $\mathbf{x}'$, to encode the codomain space, the codomain partitioning algorithm only needs one copy of the state variables $\mathbf{x}$. These variables can be recycled to encode the codomain space too. Since we will be using the codomain partitioning algorithm, henceforth we will adopt the convention of encoding the codomain or image space with the state variables. Thus, if the state variable z has a next state function n_z, then the image of function n_z over some set will be encoded by the variable z.

4.3 Computing Im_{ap} by Multiple Constrain

Recall from Section 3.2 that as we try to compute a superset (*i.e.* $FwdReach_{ap}(q_0)$) of the reachable states set, the key challenge is to compute projections of the exact images through $Im_{ap}(\mathbf{R}, \mathbf{n})$. Recall that

$$Im_{ap}(\mathbf{R}, \mathbf{n}) = (S_1, \ldots, S_p) = \alpha(Im(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))).$$

In principle, S_j can be computed through the transition relation method, by forming the next state *relation* for block w_j and using early quantification [7, 66]. However,

this did not work on the larger examples. This led us to look at ways of improvising on the transition function method of Coudert and Madre [18, 66] to compute BDDs for the S_j's efficiently.

A naive method to compute the BDDs for the various S_j's would be

1. Compute the BDD for the exact image $Im(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))$.

2. Then obtain the BDD for the various S_j's by projecting the exact image onto the various w_j subsets in $\mathbf{w}$, *i.e.* $S_j = \alpha_j(Im(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y})))$.

This naive method is not likely to succeed on practical design examples. This is because the BDD for the exact image $Im(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))$ is a BDD with a very large support set (basically all the state variables in the design) which will almost always blow up. Hence, we would like to be able to compute the S_j's separately, without computing $Im(\gamma(\mathbf{R}), \mathbf{n})$.

In fact, this is easy to achieve, and the key observation that makes this possible is that S_j can only depend on the next state functions of the variables appearing in the j^{th} block in $\mathbf{w}$, *i.e.* w_j. In our implementation, $\mathbf{n}(\mathbf{x}, \mathbf{y})$ is represented as a vector of predicates $[n_1(\mathbf{x}, \mathbf{y}), \ldots, n_k(\mathbf{x}, \mathbf{y})]$, where each predicate $n_i(\mathbf{x}, \mathbf{y})$ determines the value of state variable $\mathbf{x}_i$ in the next state. Let $\alpha_j(\mathbf{n})$ be a new vector containing only the predicates determining the next state for the bits in w_j. Clearly,

$$S_j \;=\; Im(\gamma(\mathbf{R}), \alpha_j(\mathbf{n}))$$

A naive application of the transition function method to compute the BDD for S_j would be as follows:

1. compute a BDD $P(\mathbf{x})$ by doing the explicit conjunction $\gamma(\mathbf{R})$, *i.e.* $P(\mathbf{x}) = \gamma(\mathbf{R})$.

2. Compute a new vector of functions $\alpha'_j(\mathbf{n}) = \alpha_j(\mathbf{n}) \downarrow P(\mathbf{x})$.

3. Compute the desired result through $Codomain\_Split(\alpha'_j(\mathbf{n}))$.

The very first step is a potential pitfall. Recall that the explicit conjunction $\gamma(\mathbf{R})$ would result in a big BDD with a support set over the set of state variables $\mathbf{x}$, which

can be large for practical design examples, resulting in BDD size blowup. Hence we would like to avoid computing a BDD for $P(\mathbf{x})$ through the explicit conjunction $\gamma(\mathbf{R})$.

To avoid computing the large BDD for $\gamma(\mathbf{R})$, it is tempting to do a naive *serial constrain* and instead constrain the vector of functions $\alpha_j(\mathbf{n})$ individually by the various elements of the tuple $\mathbf{R} = (R_1, \ldots, R_p)$. This would entail computing $\alpha_j(\mathbf{n}) \downarrow R_1 \downarrow R_2 \ldots \downarrow R_p$. This works well if the supports of R_i's are disjoint. (McMillan has shown [52] that **if** g and h have independent support, **then** $f \downarrow (g \wedge h) = (f \downarrow g) \downarrow h$.) However, since we have overlapping subsets, the naive method is incorrect. The following example demonstrates this.

Example 5 *Consider the functions* $f(a_1, a_2, a_3) = a_1 \wedge a_2 \wedge a_3$, $g(a_1, a_2, a_3) = a_1$ *and* $h(a_1, a_2, a_3) = \neg a_1 \vee (a_1 \wedge a_2 \wedge a_3)$. *Note that the functions g and h have overlapping support. Now* $g \wedge h = a_1 \wedge a_2 \wedge a_3$ *which is the same as f and hence* $f \downarrow (g \wedge h) = 1$ *(see properties of constrain in Section 2.5.2). However, with the naive* serial constrain *method, first* $(f \downarrow g)$ *is computed and the result is used to compute* $(f \downarrow g) \downarrow h$. *This results in the function* $a_1 \vee \bar{a}_1 a_2 a_3$, *which clearly does not match* $f \downarrow (g \wedge h)$. *(We used the variable order* $a_1 < a_2 < a_3$ *for this example.)*

Instead, for overlapping projections, we propose the following method of *multiple constrain*. The key idea behind multiple constrain is captured by the following theorem.

Theorem 5 *For any Boolean functions f, g and h, if* $g \wedge h \neq 0$, *then*

$$f \downarrow (g \wedge h) \;=\; (f \downarrow h) \downarrow (g \downarrow h)$$

Proof: *A detailed proof can be given from the definition of the constrain operator and using mathematical induction over the number of variables in support of the functions. However, a simpler proof follows from a slightly different result given by McMillan [52]. McMillan showed that for any Boolean functions f, p and q, if* $p = p \downarrow q$, *then* $f \downarrow (p \wedge q) = (f \downarrow p) \downarrow q$. *Now consider the choice of functions* $q = h$ *and* $p = g \downarrow h$. *From the properties of constrain (see Section 2.5.2), we have* $(g \downarrow h) \downarrow h = (g \downarrow h)$, *thus the requirement that* $p = p \downarrow q$ *is satisfied. Further for*

this choice of functions, $p \wedge q = (g \downarrow h) \wedge h = g \wedge h$. Applying McMillan's result for this choice of functions, we have $f \downarrow (g \wedge h) = (f \downarrow h) \downarrow (g \downarrow h)$, which completes the proof.

Theorem 5 allows constraining a function with the *implicit* conjunction of *two* other functions. It can be further extended to allow for constraining a function by the implicit conjunction of a *list* of functions.

Corollary 1 *For any Boolean function f, and a list of functions $(R_1, \ldots, R_p)$, if $\wedge_{i=1}^{p} R_i \neq 0$ then*

$$f \downarrow (R_1 \wedge R_2 \wedge \ldots \wedge R_p) \;=\; (f \downarrow R_p) \downarrow ((R_1 \wedge \ldots \wedge R_{p-1}) \downarrow R_p).$$

To use this decomposition recursively, we can use the additional distributive property of constrain, *i.e.*

$$(R_1 \wedge \ldots \wedge R_{p-1}) \downarrow R_p = (R_1 \downarrow R_p) \wedge \ldots \wedge (R_{p-1} \downarrow R_p).$$

Interestingly enough, the generalized result of Theorem 5 which is directly applicable for our purposes, was actually inspired by an earlier less-general observation that the range of $f \downarrow (g \wedge h)$ is the same as the range of $(f \downarrow h) \downarrow (g \downarrow h)$. (The reader can choose to skip the remaining part of this section and go to Section 4.3.1 directly without any loss of continuity.)

This less-general observation is based on the following key relation between image computation and range computation. Let $(z_1, \ldots, z_p)$ be dummy state bits with corresponding next state functions $(R_1, \ldots, R_p)$, then

$$Im(\gamma(R_1, \ldots, R_p), \alpha_j(\mathbf{n})) =$$
$$Im(1, [\alpha_j(\mathbf{n}), R_1, \ldots, R_p]) \downarrow z_1 \downarrow z_2 \ldots \downarrow z_p.$$

In other words, we first extend the Boolean function vector $\alpha_j(\mathbf{n})$ with $(R_1, \ldots, R_p)$, and compute the range of the extended vector to get the set of next states. Every point in the range of $[\alpha_j(\mathbf{n}), R_1, \ldots, R_p]$ will be "tagged" with the dummy variables z_i, which keeps track of the R_i's that were satisfied in the present state. The required

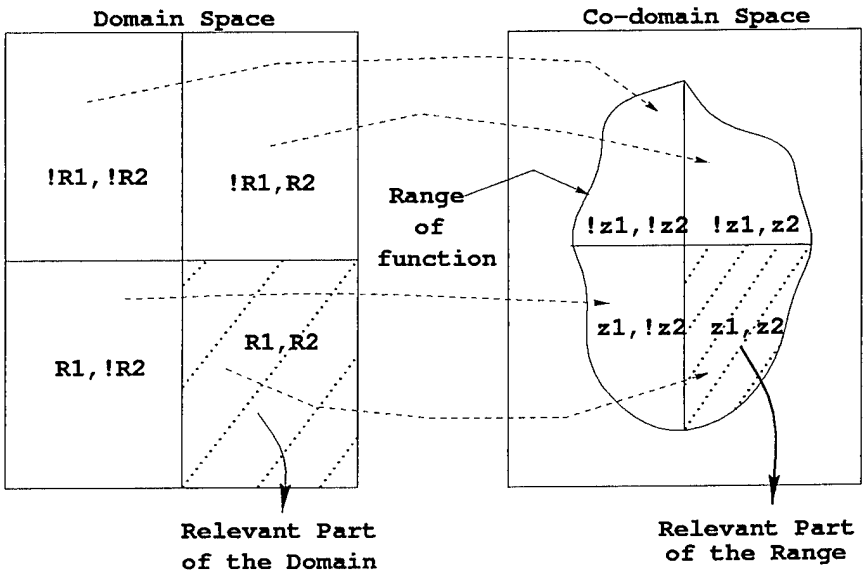


Figure 4.1: Intuition to multiple constrain

image is the part of the range where all the dummy bits $(z_1, \ldots, z_p)$ are 1 (*i.e.* where all the R_i's were satisfied by the present state). Figure 4.1 captures this idea. Selecting the cofactors where $z_1 = z_2 = \ldots = z_p = 1$ finds the BDD for the relevant part of the range while eliminating the dummy z_i variables. Continuing from Example 5, the *multiple constrain* technique would compute $(f \downarrow h) \downarrow (g \downarrow h) = 1$. Hence, we get $Im(1, (f \downarrow h) \downarrow (g \downarrow h)) = z$ which matches the required correct result since $Im(1, (f \downarrow (g \wedge h))) = z$.

We can optimize on the usual recursive co-domain partitioning algorithm [18], by avoiding the computation of the parts of the range that will be discarded. In order to achieve this, we start with the augmented vector of predicates, $[\alpha_j(\mathbf{n}), R_1, \ldots, R_p]$, and constrain each element of the vector with R_p (the last element of the vector). The process is repeated again where each element of the new vector is constrained by the new R_{p-1}. This process of constraining every element of the vector by the next constrained R_i is repeated p times until the elements of the vector are constrained by the final R_1. Thereafter, we do the standard co-domain recursive range computation through the *Codomain_Split* algorithm (as given in Section 4.2.2). The final algorithm is defined more formally in Section 4.3.1.

4.3.1 Multiple Constrain Algorithm

Using the result of Corollary 1, we can formally define the algorithm for computing the image of an implicit conjunction of BDDs.

function $Im_{mc}((R_1, \ldots, R_p), [n_1, \ldots, n_m])$
$\mathbf{v} \leftarrow [n_1, \ldots, n_m, R_1, \ldots, R_p]$
for j=p **down to** *1* **by** *1* **do**
  if $(v[m+j] == 0)$
    print "domain empty" as $\gamma(\mathbf{R}) = 0$
    return 0
  else
    $\mathbf{v} \leftarrow \mathbf{v} \downarrow v[m+j]$
endfor
return $Codomain\_Split([v[1], \ldots, v[m]])$

Our least fix-point routine starts with $\mathbf{R}$: $(0, \ldots, 0)$ and computes the tuple $Reach_{ap}$ as,

$$\text{lfp } \mathbf{R}.(\alpha(q_0) \sqcup (Im_{mc}(\mathbf{R}, \alpha_1(\mathbf{n})), \ldots, Im_{mc}(\mathbf{R}, \alpha_p(\mathbf{n}))$$

Our algorithm most closely resembles the RFBF algorithm proposed by Cho *et al.* [12], but differs in that we allow for overlapping projections and compute the image for each block with our new Im_{mc} operator. It is also straightforward to do MBM, TFBF, TMBM [12], LMBM [54] traversals using overlapping projections.

4.4 Optimizations

BDD-based algorithms rely on memo-ization heavily to get efficiency. In the algorithm *Codomain_Split*, the cache stores entries where the tags are vectors of functions and the data is the corresponding image BDD. Getting cache hits becomes a low probability event since now we need a vector of functions to match. Several optimizations

have been proposed [18] to help increase the cache hit rate and reduce the number of recursions.

The *Codomain_Split* algorithm takes $\mathbf{f_n} = [f_1, \ldots, f_n]$ as an argument, where each f_i is the next state function for some state variable x_i. Another simple optimization is to sort the functions in the vector $\mathbf{f_n}$ so that f_i's in the vector appear in the same order as the x_i's appearing in the global variable order. Thus, if $x_i < x_j$ in the variable order, then f_i should appear before f_j in the vector $\mathbf{f_n}$. As a result, the BDD that is being recursively created by *Codomain_Split* is aligned with the variable order at all times.

4.5 Choosing the Collection of Subsets

The quality of approximation is highly dependent on the choice of subsets. The key intuition is that since interaction and correlation between state bits in different subsets is lost, we should try to ensure that state bits which are highly dependent or correlated with other state bits, should be in the same subset as the bits they depend on or correlate to.

4.5.1 Leveraging High level Information

For the design examples from the MAGIC [45] chip, we had access to the high level design description in Verilog [63]. This enabled leveraging off some high level information that can be deduced by inspecting the design description. Our scheme for choosing the collection of subsets is presently manual.

First, we find the FSMs by inspecting the HDL source (we had access to the RTL description for our design examples). For each state bit x_i, a score is computed by counting the number of predicates $n_j(\mathbf{x}, \mathbf{y})$ it supports. To each machine, a score is assigned which is the sum of the scores of its state bits. The two machines (M_1, M_2) with the highest scores are identified as *master* FSMs. If the state bits of machines M_i and M_j support the bits of the *master* machine M_1 in their next state predicates, then M_i and M_j are *slaves* of M_1. The different slave machines for each of the master

FSMs are identified. Blocks are formed by pairing the master FSMs with their slaves. Thus, in this case, the blocks (M_1, M_i) and (M_1, M_j) are added to the collection of subsets.

Often some FSMs are very small. The corresponding small blocks can then be aggregated with other blocks without running into intermediate image BDD size explosion. The converse problem is some FSM, say M_i, may have large state registers, resulting in big blocks. If so, we try to prune these blocks by exploiting the *small interface* phenomenon, described in Section 3.2.2. A block with the master FSMs is also added, to capture the *correlation* between the FSMs. We ensure that no block w_i in the collection $\mathbf{w}$ is a proper subset of another block $w_j \in \mathbf{w}$, since this would clearly be wasteful.

4.5.2 Structural Methods for Gate Level Net-lists

Cho *et al.* [11] proposed several heuristics to choose the collection of subsets when a design is available in a gate level net-list form (as is the case with the ISCAS-89 benchmark suite). They proposed different metrics to quantify the dependence and correlation between different state variables in a net-list. We use the same choice of subsets as given by their heuristics as an initial starting point, and then add more bits to allow for overlap.

The heuristic to add more overlap bits on top of the choice of subsets from Cho *et al.* [12] is as follows. Let $(x_1, \ldots, x_{10})$ be ten different state variables in a block, and let $(n_1, \ldots, n_{10})$ be the associated next state functions for the state bits in the block. The state variable x_j (where $x_j \notin \{x_1, \ldots, x_{10}\}$), which appears in the support of most of the $(n_1, \ldots, n_{10})$ functions, is identified. The subset $(x_1, \ldots, x_{10})$ is then augmented by adding the state variable x_j to it. This simple heuristic tries to add more bits to capture the dependence interaction between state bits, and has helped to improve the quality of approximation by orders of magnitude.

4.6 Experimental Results

The experimental implementation of the method was in LISP, calling David Long's BDD package [48] (implemented in C) via the foreign function interface. The method was evaluated on a collection of control circuits from the MAGIC chip, a custom node controller in the Stanford FLASH multiprocessor [45]. For comparison with earlier work, results when applied to the publicly available ISCAS89 benchmark circuits are also presented. The approximate algorithm returns a superset of the reachable states, which is also an invariant of the design. To quantify the size of the superset, the satisfying fraction of the the superset is computed. (Please refer to the Appendix of this chapter in Section 4.8 for the algorithm that was used to compute an upper bound on the satisfying fraction). Since projection induces an over-approximation, the smaller the satisfying fraction, the stronger the invariant.

The maximum number of BDD nodes (BDD Node Limit) for each experiment (*i.e.* for each row in the following tables) was preset. Initially the collection of subsets, $\mathbf{w}$, has small sized *disjoint* subsets and these subsets incrementally become larger, until the experiment requires more BDD nodes than set in the limit. To this collection of disjoint subsets giving the best result within the node limit, small overlap bits were added as per the heuristics given earlier (Section 4.5). Thus, by staying within the bounds of the node limit, the strongest invariant obtained with overlapping projections is compared to the strongest invariant obtained with disjoint partitions. The column *Subsets* lists different choices of the collection of subsets, $\mathbf{w}$, where the size of subsets increases as we go down a table. The same variable ordering was used for both the schemes.

Nodes keeps track of the peak number of nodes that existed at a time during the experiment. The *Iter* column lists the number of iterations needed to reach the fix-point. The last column under the heading *Ratio* is the ratio between the satisfying fraction with disjoint partitions and the satisfying fraction with overlapping projections. Thus, the higher the figures in the *Ratio* column, the stronger is the invariant with overlapping projections.

4.6.1 Results on Design Examples from FLASH

The following tables summarize the results obtained on the various FLASH I/O modules (in the order IOInboxQCtl, ReqDecode, ReqService, IOMiscBusCtl and PciInterface) as we compare earlier schemes of approximations by disjoint partitions [12] with the new scheme of approximations by overlapping projections. Note that for a given node limit budget, overlapping projections consistently result in tighter approximations than their disjoint partition counterparts.

Table 4.1: IOInboxQCtl design example results

Subsets	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
$\mathbf{w}_1$	5.005e-03	20	28254	5.005e-03	20	28254	1.000
$\mathbf{w}_2$	"	"	"	4.944e-03	20	53740	1.012
$\mathbf{w}_3$	"	"	"	3.967-03	20	64462	1.262
$\mathbf{w}_4$	3.967e-03	20	76630	3.967e-03	20	64462	1.000

Table 4.1: Note that to improve upon the invariant with satisfying fraction 5.005e-03, in the case of disjoint partitions, the BDD node count had to jump from 28,254 to 76,630. which is a 2.71 times increase in the BDD node count. The last entry under disjoint partitions was computed with all the state variables in a single block, which clearly gives the strongest possible invariant. Overlapping projections produces this same strong invariant at a lower node count.

Table 4.2: ReqDecode design example results

Subsets	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
$\mathbf{w}_1$	2.185e-05	20	33408	2.185e-05	20	33408	1.000
$\mathbf{w}_2$	2.108e-05	20	134536	1.979e-05	20	171448	1.065
$\mathbf{w}_3$	1.274e-05	33	980968	1.018e-05	20	608726	1.252
$\mathbf{w}_4$	"	"	"	8.156e-06	20	1195109	1.562
$\mathbf{w}_5$	3.169e-06	25	2032890	3.169e-06	25	2032890	1.000

Table 4.2: For the choice of subsets, $\mathbf{w}_3$ and $\mathbf{w}_4$, the algorithm with overlapping projections yields stronger invariants (by a factor of 1.252 and 1.562 respectively).

Furthermore, for the choice of subsets $\mathbf{w}_3$, the algorithm with overlapping projections uses fewer BDD nodes compared to the RFBF runs with disjoint partitions, and at the same time gives a stronger invariant.

Table 4.3: ReqService design example results

Subsets	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
$\mathbf{w}_1$	1.658e-02	34	23662	1.658e-02	34	23662	1.000
$\mathbf{w}_2$	1.352e-03	44	407728	1.053e-03	37	470642	1.283
$\mathbf{w}_3$	"	"	"	1.039e-03	40	537578	1.300
$\mathbf{w}_4$	"	"	"	1.039e-03	40	1776965	1.300
$\mathbf{w}_5$	"	"	"	1.036e-03	44	1995305	1.304
$\mathbf{w}_6$	1.036e-03	44	11007330	"	"	"	1.000

Table 4.3: With disjoint partitions, the node count penalty goes up from 407,728 to 11,007,330 (a factor of 27) before any improvement in the strength of the invariant. The last entry under disjoint partitions was computed with all state variables in a single block, which gives the strongest invariant. Note that the same invariant is obtained by the overlapping projections scheme at a much lower node count penalty (1,995,304 nodes *vs* 11,007,330 nodes, which is lower by a factor of 5.517).

Table 4.4: IOMiscBusCtl design example results

Subsets	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
$\mathbf{w}_1$	4.211e-04	4	104135	4.211e-04	4	104135	1.000
$\mathbf{w}_2$	3.810e-04	4	1173863	2.727e-05	4	1244294	13.973
$\mathbf{w}_3$	"	"	"	5.342e-06	4	1353024	71.329
$\mathbf{w}_4$	5.342e-06	4	2556733	"	"	"	1.000

Table 4.4: Note that for the choice of subsets $\mathbf{w}_2$ and $\mathbf{w}_3$, the algorithm with overlapping projections yields significantly stronger invariants (by a factor of 13.973 and 71.329 respectively), at only an incremental additional cost in terms of BDD node count. Figure 4.2 is a plot of the satisfying fraction of the final result *vs* the peak number of BDD nodes. Since a lower satisfying fraction implies a stronger invariant,

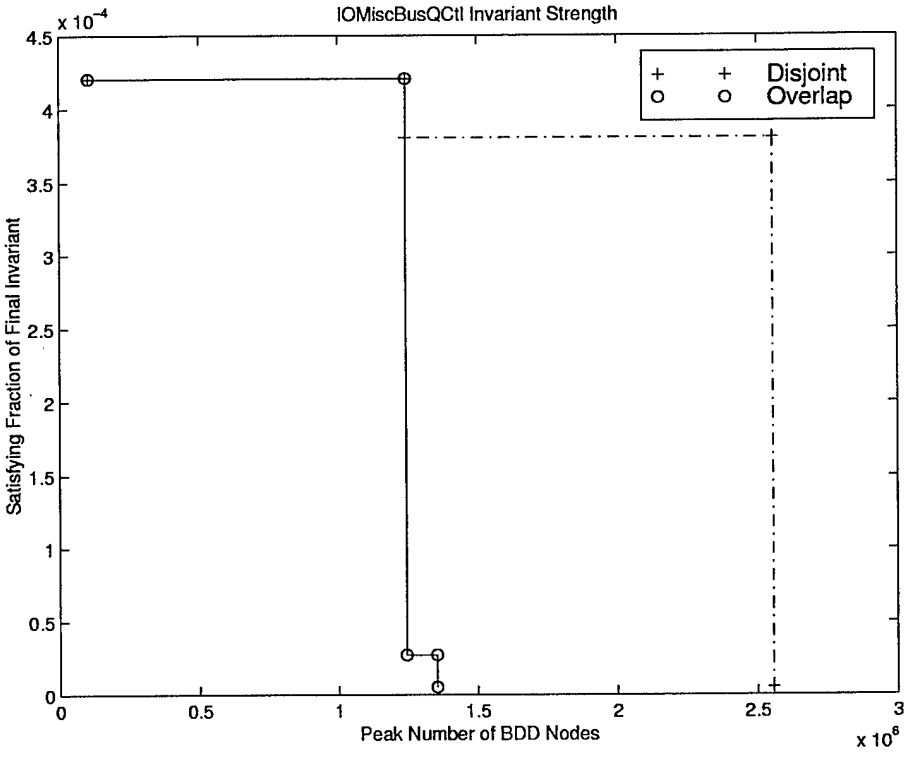


Figure 4.2: IOMiscBusCtl: Projections *vs* Partitions

it is expected that lower satisfying fractions would incur a higher BDD node count penalty. The solid curve for overlapping projections is considerably below the other curve for disjoint partitions, indicating that overlapping projections give stronger results with lower BDD node counts compared to disjoint partitions.

Table 4.5: PciInterface design example results

Subsets	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
$\mathbf{w}_1$	3.574e-03	21	283186	1.168e-04	50	228390	30.593
$\mathbf{w}_2$	1.041e-05	55	598257	1.041e-05	55	598257	1.000
$\mathbf{w}_3$	9.463e-07	71	1616055	6.311e-07	71	1293062	1.499
$\mathbf{w}_4$	"	"	"	1.595e-07	71	20851528	5.932

Table 4.5: The final result with overlapping partitions is much stronger (by a factor of 5.932) than that obtained with disjoint partitions. Note that even as the size of the subsets increases from $\mathbf{w}_3$ to $\mathbf{w}_4$, there is no improvement in the disjoint partition case. (The choice of subsets $\mathbf{w}_3$ was relative to a node limit of 2 million nodes, while the choice of subsets $\mathbf{w}_4$ was relative to a node limit of 25 million nodes. Thus, even as the node limit is raised significantly, no improvement of the result is obtained by using the disjoint partition based scheme.)

4.6.2 Results on ISCAS-89 Benchmark Circuits

Table 4.6: Large circuits from ISCAS-89 benchmark suite

Circuit	State Bits	Input Bits
s1423	74	17
s13207	669	31
s15850	597	14
s38584	1452	12

The algorithm was also evaluated on the bigger benchmarks in ISCAS 89 benchmark suite. Table 4.6 gives some information on the size of these circuits. The partitions used by Cho *et al.* [12] were used to identify the FSMs in the design. For

the overlapping projections case, variable subsets were set (as per the heuristic given in Section 4.5) by adding small overlaps to some of their blocks. We are unable to report comparative figures for s35932, because we could not procure the partitions used by Cho *et al.* for s35932. In the case of of s5378, our version of s5378 had 179 flip flops as opposed to the 164 flip flops in the one used by Cho et al. Table 4.7 has details of the results on the remaining benchmarks. Note that there is *orders of magnitude improvement* in the strength of the invariant for s13207 and s38584. The numbers in Table 4.7 under overlapping projections are *upper bound estimates* of the satisfying fraction of the final invariant. Thus, the invariant with overlapping projections is stronger, *at least* by a factor equal to the figures under the *Ratio* column.

Table 4.7: ISCAS 89 benchmarks: Size of approximate forward reachable set

Ckt	Disjoint Partitions			Overlapping Projections			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
s1423	2.985e-03	37	310461	2.193e-03	248	1032286	1.361
s13207	3.421e-106	10+6	161447	1.136e-115	10+5	198779	3.321e+08
s15850	5.840e-102	10+5	271093	3.938e-102	10+4	336048	1.483
s38584	6.494e-41	10+2	646258	5.764e-57	10+5	1853461	8.876e+15

The numbers under the *Disjoint Partitions* column correspond to the results obtained by running TMBM [12] approximate traversal algorithm for the circuits: s13207, s15850, s38584, and RFBF [12] approximate traversal algorithm for the circuit: s1423. The same partitions used by Cho *et al.* [12] were used here. The TMBM traversal algorithm starts off as TFBF and switches to MBM after a few iterations. Since we are using TMBM algorithm for some circuits, the *Iter* column in Table 4.7, lists the number of iterations of doing TFBF + the number of iterations in the outer greatest fix-point of MBM.

4.7 Conclusions

Overlapping projections has proved to be a very effective approximation scheme. It has enabled orders of magnitude improvement in the size of the superset returned

by the approximate analysis, when compared against earlier approximation schemes. Our experiments show that a small amount of appropriately chosen overlaps in a given projection can substantially improve the quality of the over-approximation.

Multiple constrain has proved to be an efficient method to compute the image of an *implicit* conjunction of BDDs with possible overlapping support, using Boolean function vectors.

4.8 Appendix

4.8.1 Approximating *Sat_Fr* of Superset

The approximate least fix-point routine returns a list of BDDs, $\mathbf{S} : (S_1, \ldots, S_p)$ corresponding to the collection $\mathbf{w} : (w_1, \ldots, w_p)$. The *implicit* conjunction of the BDDs in the list $\mathbf{S}$ represents a superset of the reachable state space. In order to quantify the number of states in the superset, the satisfying fraction of $\gamma(\mathbf{S})$ is to be computed. Schemes that rely on first building the BDD for $\gamma(\mathbf{S})$ and then computing the satisfying fraction of the resulting BDD will usually fail, because it is prohibitively expensive to build a BDD with the explicit conjunction through γ.

To the best of our knowledge, computing the exact satisfying fraction of $\gamma(\mathbf{S})$ by merely manipulating the individual BDDs in the list $\mathbf{S}$ is not possible. (If elements of $\mathbf{S}$ had mutually disjoint support, multiplying the satisfying fractions of the individual BDDs in the list would suffice. This is because, under the assumption of mutually disjoint support sets of the various BDDs in the list $\mathbf{S}$, the following result holds: $\prod_{i=1}^{p} sat\_fr(S_i) = sat\_fr(\wedge_{i=1}^{p} S_i)$.)

But in this scheme of approximation with overlapping projections, different BDDs in the list $\mathbf{S}$ are expected to have overlapping support. Hence, we settle for an algorithm that compute an upper bound on *sat_fr* of $\gamma(\mathbf{S})$. The greedy algorithm defined below achieves this by computing *sat_fr* of a superset of $\gamma(\mathbf{S})$. It uses the fact that $\exists a.(f \wedge g) \subseteq (\exists a.f) \wedge (\exists a.g)$,

A set Z is used to keep track of the variables to hide existentially, before computing *sat_fr* of each block. At every step the BDD S_m, with the lowest *sat_fr* (after hiding

existentially variables in Z from S_m), is picked. Its *sat_fr* is cumulatively multiplied to f, and variables in w_m are added to the set Z.

$Z \leftarrow \emptyset$; $f \leftarrow 1.0$
for *j=1* **up to** p **by** *1* **do**
find m, s.t. $\forall i.(sat\_fr(\exists Z.S_m) \leq sat\_fr(\exists Z.S_i))$
$f \leftarrow f \times sat\_fr(\exists Z.S_m)$
$Z \leftarrow Z \cup w_m$
endfor
return f

The Monte Carlo simulation technique, an alternative method to estimate the satisfying fraction of $\gamma(\mathbf{S})$, appears to be ineffective because of the extreme sparseness of the state space covered by $\gamma(\mathbf{S})$. To get estimates with a good confidence interval, a prohibitively large number of samples would be needed.

Chapter 5

Approximate Backward Reachability

The general himself ought to be such a one as can at the same time see both forward and backward. — Plutarch.

This chapter first defines the key challenge in symbolic backward reachability, namely the pre-image computation problem. Existing methods of BDD based pre-image computation are briefly reviewed. A new pre-image computation method, which allows for efficient computation of projections of exact pre-images, is defined. This method of backward reachability is combined with the forward reachability method of the previous chapter. A simple heuristic to generate counterexample paths (from the initial state to the error states) from the resulting approximations is presented. Finally, the results obtained by applying this method to different design examples are presented.

5.1 Basic Algorithm

Computing the set of states that can reach certain states is a key part of model checking. In Section 2.3, we saw how BDDs can be used to represent Boolean functions,

sets of states, and relations. This enables the modeling of synchronous hardware designs with BDDs.

Let g be a set of states that satisfies a user specified property (represented by the BDD $g(\mathbf{x})$). The BDD $\neg g(\mathbf{x})$ represents the set of states that violates the user specified property. It is important to know if any of the states in the set, represented by $\neg g(\mathbf{x})$, are reachable from the initial states.

We wish to compute a BDD $R(\mathbf{x})$ that represents the states that can reach the error states $\neg g$ via the transition relation T (represented by the BDD $T(\mathbf{x}, \mathbf{x}')$). We first consider the problem of finding those states R_1, that can reach $\neg g$ in at most one step. This set of states is given by

$$R_1 = \neg g \cup \{s \mid \exists s' \cdot [s' \in \neg g \wedge (s, s') \in T]\}$$

Given the BDDs $\neg g(\mathbf{x})$ and $T(\mathbf{x}, \mathbf{x}')$, we can compute a BDD representing R_1 by performing the logical operations corresponding to the above expression:

$$R_1(\mathbf{x}) = \neg g(\mathbf{x}) \vee \exists \mathbf{x}' \cdot [\neg g(\mathbf{x}') \wedge T(\mathbf{x}, \mathbf{x}')].$$

Similarly, the set of states that can reach $\neg g$ in at most two steps is represented by

$$R_2(\mathbf{x}) = \neg g(\mathbf{x}) \vee \exists \mathbf{x}' \cdot [R_1(\mathbf{x}') \wedge T(\mathbf{x}, \mathbf{x}')].$$

In general, the set of states that can reach $\neg g$ in at most $n + 1$ steps is represented by

$$R_{n+1}(\mathbf{x}) = \neg g(\mathbf{x}) \vee \exists \mathbf{x}' \cdot [R_n(\mathbf{x}') \wedge T(\mathbf{x}, \mathbf{x}')].$$

Note that each set of states is a superset of the previous one. Since the total number of states in a hardware design is finite, at some point we must have $R_{n+1} = R_n$. No further states can possibly reach the error states $\neg g$. If the initial states set q_0 does not intersect with the set R_n, then we can safely conclude that the error states are

definitely not reachable from the initial states. On the other hand, if the initial states set q_0 does intersect with the set R_n, it means there exists a *counterexample* path starting from the initial states and ending in the error states $\neg g$. Such a counterexample path has immense debugging value, since a designer can inspect it and exactly locate the problem.

5.2 Methods to Compute Pre-images

The key step in the high level algorithm outlined earlier is computing the one step predecessors of a set of states. This is widely referred to as *pre-image* computation. Existing methods of BDD based pre-image computation can be broadly classified into two categories.

5.2.1 Transition Relation Approach

As outlined in the previous section, this method relies on building BDDs to represent the transition relation $T(\mathbf{x}, \mathbf{x}')$ of the circuit. The key problem is computation of the relational product:

$$R(\mathbf{x}) \;=\; \exists \mathbf{x}' \cdot [R'(\mathbf{x}') \wedge T(\mathbf{x}, \mathbf{x}')].$$

A close look reveals that the pre-image computation, using the transition relation, is symmetrical to the image-computation problem we saw in Section 4.2.1. The only difference is that instead of quantifying out the $\mathbf{x}$ (present state variables) variables, pre-image computation involves quantifying out the $\mathbf{x}'$ (next state versions of the state variables) variables.

Just as in the image computation case using transition relations, the pre-image can be computed using the normal BDD algorithms for restriction and Boolean connectives. However, it does not work well in practice for large designs. This is because the basic algorithm requires having $T(\mathbf{x}, \mathbf{x}')$ as a monolithic relation, consisting of a single BDD. Unfortunately, for most practical designs, this BDD is very large. It is much more efficient to use a special purpose algorithm, based on partitioned transition

relations. The problem is now of the form

$$R(\mathbf{x}) = \exists \mathbf{x}' \cdot [R'(\mathbf{x}') \wedge (t_1(\mathbf{x}, x_1') \wedge t_2(\mathbf{x}, x_2') \wedge \ldots \wedge t_n(\mathbf{x}, x_n'))].$$

The main difficulty in computing $R(\mathbf{x})$ without building the conjunction is that existential quantification does not distribute over conjunction.

Similar to the case of image computation using transition relations, the BDD $R(\mathbf{x})$ for the pre-image can be efficiently computed using early quantification optimizations (Section 4.2.1).

5.2.2 Function Substitution Approach

For deterministic systems, Filkorn [26] proposed an alternative to the transition *relations* based method of computing pre-images [7, 47]. Filkorn [26] showed that the pre-image of a set, represented by a BDD $Q(\mathbf{x})$, can be obtained by substituting the state variables in $Q(\mathbf{x})$ with their corresponding next state *functions.*

BDD packages usually have support for a *compose* operation, whereby a variable in a BDD can be substituted with a function. In the functional substitution approach, the computation of the pre-image of a set represented by the BDD $Q(\mathbf{x})$, is done as follows:

1. Rename the $\mathbf{x}$ variables in $Q(\mathbf{x})$ to their primed versions to obtain $Q(\mathbf{x}')$

2. for each x_i' in the support of $Q(\mathbf{x}')$, substitute the next state function of x_i for each occurrence of x_i' in $Q(\mathbf{x}')$.

3. existentially quantify out the input variables from the resulting BDD to obtain the required pre-image.

Filkorn [26] argued that in the case of deterministic synchronous digital circuits, a functional instead of a relational representation results in more compact BDDs.

5.3 Computing Pre_{ap} by Domain Cofactoring

Now let us try to apply the function substitution method to our applications. Recall from Section 3.2 that as we try to compute a superset (*i.e.* $BackReach_{ap}(\neg g)$) of the states that can reach the error states, the key challenge is to compute projections of the exact pre-images through $Pre_{ap}(\mathbf{R}, \mathbf{n})$. Recall that

$$Pre_{ap}(\mathbf{R}, \mathbf{n}) = (S_1, \ldots, S_p) = \alpha(Pre(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))).$$

In principle, S_j can be computed through the transition relation method, by forming the next state *relation* for block w_j and using early quantification [7, 66]. However, this did not work when we tried it on larger examples. This led us to look at ways to improvise with the function substitution method of Filkorn [26], to compute BDDs for the S_j's efficiently.

A naive method to compute the BDDs for the various S_j's would be

1. Compute the BDD for the exact pre-image $Pre(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))$. This could be done by either the transition relation based method or by the function substitution method.

2. Then obtain the BDD for the various S_j's by projecting the exact pre-image onto the various w_j subsets in $\mathbf{w}$, *i.e.* $S_j = \alpha_j(Pre(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y})))$.

This naive method is not likely to succeed on practical design examples. This is because the BDD for the exact pre-image $Pre(\gamma(\mathbf{R}), \mathbf{n}(\mathbf{x}, \mathbf{y}))$ has a very large support set (basically all of the state variables in the design), which will almost always lead to BDD size blow up. Even though the final BDD for S_j is expected to be small because of its restricted support within w_j, this method requires us to pay the prohibitively expensive price of first building a big BDD for $Pre(\gamma(\mathbf{R}), \mathbf{n})$ and then hiding some variables to get the smaller sized BDDs for S_j. It is important to be able to compute the S_j's separately without computing $Pre(\gamma(\mathbf{R}), \mathbf{n})$.

Our method for computing S_j involves recursively splitting the domain variables in w_j. This not only allows us to directly compute S_j without computing the exact

pre-image, but also allows the existential quantification to be done on the fly. As we split on a variable v in w_j, we create two subproblems. After splitting on all the variables in w_j we have created a large number of subproblems, but each of these subproblems can be solved easily. This is because after fixing the values of all the state variables in w_j, all the functions typically get simplified to small BDDs with support outside w_j. At this point, the BDD we are constructing for S_j depends only on whether the substitution of functions in $\gamma(\mathbf{R})$ results in a satisfiable function. If so, we return 1 for the recursive BDD computation, otherwise we return 0.

Initially each state variable x_i in $\mathbf{R}$ is renamed to x_i' to avoid conflicts. Let σ be a map from each x_i' to the function that is to be substituted for it. Initially, σ maps x_i' to the next state function of x_i. As the splitting process starts, σ gets modified. Once a function from σ is substituted in the R_i's, it is removed from σ. Thus, if we let $|\sigma|$ denote the number of functions in σ left to be substituted, this can only decrease as the recursion unfolds, and we use this to define the terminal case of the recursion.

The recursive algorithm Pre_{dc} (the subscript dc denotes "domain cofactoring") takes as arguments the current substitution, σ, the current approximation $\mathbf{R}$, the approximate reachability set from the previous forward pass $\mathbf{I}$, and the set of variables w_j to project onto. $\mathbf{I}$ is used to prune pre-image states that are definitely not reachable. (The algorithm shown below to compute S_j, assumes there are only two subsets in our collection $\mathbf{w}$. The extension to an arbitrary number of subsets is obvious. We use $\downarrow$ to denote the ordinary cofactor operator.)

function $Pre_{dc}(\sigma, [R_1, \ldots, R_p], [I_1, \ldots, I_p], w_j)$
  if $((I_1 == 0)$ **or** $\ldots$ **or** $(I_p == 0))$ **return** 0
  if $(|\sigma| == 0)$ **return** $R_1 \wedge R_2 \wedge \ldots \wedge R_p$
  $v \leftarrow$ *next variable from* w_j *to cofactor on*
  $t \leftarrow Pre_{dc}(\sigma \downarrow_v, [R_1 \downarrow_v, \ldots, R_p \downarrow_v], [I_1 \downarrow_v, \ldots, I_p \downarrow_v], w_j)$
  $e \leftarrow Pre_{dc}(\sigma \downarrow_{\bar{v}}, [R_1 \downarrow_{\bar{v}}, \ldots, R_2 \downarrow_{\bar{v}}], [I_1 \downarrow_{\bar{v}}, \ldots, I_2 \downarrow_{\bar{v}}], w_j)$
  result $\leftarrow ite(v, t, e)$
return *result*

The following optimizations reduce the number of recursive calls to Pre_{dc}:

- We use the invariant generated by approximate forward reachability, *i.e.* $\mathbf{I} = (I_1, I_2)$, to prevent the approximate pre-images from including states that are definitely unreachable. Notice that we split the elements of the tuple $\mathbf{I}$ at each step, and return 0 if any element of $\mathbf{I}$ reaches its 0 node. This results in an *on-the-fly* conjunction of the approximate pre-image with the invariant. In our experience, this significantly reduces the number of recursion steps. The fact that overlapping projections result in tight over-approximations helps to prune away many unnecessary recursion steps.

- The substitution σ only includes functions that need to be substituted into the R_i's. Further, if at any point the support of a function in σ is wholly contained inside w_j, it is immediately substituted into the R_i's and thereafter removed from σ. When $|\sigma| = 0$, all the the support of all R_i's is contained in w_j, so the algorithm computes their explicit conjunction and returns.

  Note that because of this "early substitution" of some functions, the variables in $\mathbf{R}$ are initially renamed to their next state versions. This ensures that we do safe substitution and that the various substitutions don't interfere with each other. As the recursion unfolds, the support of the R_i's starts including both the present state variables $\mathbf{x}$ and their next state versions $\mathbf{x}'$. Because of the possible presence of the present state variables in $\mathbf{R}$ we need to cofactor the R_i's with the splitting variable v for the subsequent recursive steps.

  Thus, if the recursion gets to a stage where all the functions in σ have support within $\{w_j \cup \mathbf{y}\}$, then all functions in σ are substituted in the various $R_i's$. Then the $R_i's$ are explicitly conjuncted and *after* that the inputs are existentially quantified. This step almost suggests that the algorithm finds the pre-images of individual R_i's and then conjoins the individual pre-images. Even though, in general, pre-images of relations do not distribute over conjunctions, this step is justified in our case. This is because the underlying model here is *deterministic* Mealy machines (which have next state *functions*), and because the inputs are

removed at the end. Please refer to the Appendix of this chapter for a formal proof of this claim.

- The algorithm splits only on the variables in w_j. Before choosing the next variable from w_j to split upon, we make sure that the variable v appears in the support of some function in σ. If it doesn't, we skip it and try the next candidate from w_j.

- After cofactoring on variables in w_j, the support of the functions in σ is disjoint from w_j, and now the result of Pre_{dc} is either 0 or 1. At this point, we need to check if there is some assignment to variables in $(\mathbf{x} - w_j)$ that lies in the pre-image of $\gamma(\mathbf{R})$, where $\mathbf{R} : (R_1(w_1'), \ldots, R_p(w_p'))$. Let $T_\sigma(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ denote the *implicitly* conjoined transition relation, $T_\sigma(\mathbf{x}, \mathbf{y}, \mathbf{x}') = \wedge_{j=1}^{k}(x_j' \equiv n_j(\mathbf{x}, \mathbf{y}))$, where $n_j(\mathbf{x}, \mathbf{y})$ is obtained from σ. (Note that since we have already cofactored on all of the variables in w_j, at this point the support of $T_\sigma(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ includes only those state bits not in w_j).

  We are interested in knowing if $\exists \mathbf{x}'(R_1(w_1') \wedge \ldots \wedge R_p(w_p') \wedge T_\sigma(\mathbf{x}, \mathbf{y}, \mathbf{x}'))$ is 0 or not. Clearly, this can be reduced to checking if $(R_1(w_1') \wedge \ldots \wedge R_p(w_p') \wedge T_\sigma(\mathbf{x}, \mathbf{y}, \mathbf{x}'))$ is 0 or not. Please see the Appendix of this chapter to see how the multiple constrain operator is used in a novel fashion to solve this satisfiability problem of an implicit conjunction of BDDs.

  This approach worked fine on all the examples that were tested; however, in case of BDD blowup, the algorithm could return a conservative value of 1.

5.4 Combining Forward/Backward Reachability

Using Im_{mc} and Pre_{dc}, we compute an over-approximation of the set of states that are on a path from the initial state to an error state. If the traversals were exact, this set of states could be computed by a single forward and backward pass. However, since the images and pre-images are approximate, additional passes may increase the accuracy of the result. In more detail, each step of each forward and backward

traversal is intersected with the set of states computed by the previous traversals. We alternate forward and backward traversals until the set of states no longer changes.

Given a designer provided invariant $g(\mathbf{x})$, we will use g to denote the "good" states and $\neg g$ to denote the "error" states. The algorithm for the iterative refinement technique can now be described as follows:

function *BackAndForth* (g)
$\mathbf{R_f} \leftarrow (0, \ldots, 0)$
$\mathbf{R_b} \leftarrow (1, \ldots, 1)$
while $(\mathbf{R_f} \neq \mathbf{R_b})$ **do**
  $\mathbf{R_f} \leftarrow$ *lfp* $\mathbf{R}.(\alpha(q_0) \sqcup (Im_{ap}(\mathbf{R}, \mathbf{n}) \sqcap \mathbf{R_b}))$
  if $(\gamma(\mathbf{R_f}) \rightarrow g)$ **return** *"no errors"*
  $\mathbf{R_b} \leftarrow$ *lfp* $\mathbf{R}.(\alpha(\neg g) \sqcup (Pre_{ap}(\mathbf{R}, \mathbf{n}) \sqcap \mathbf{R_f}))$
  if $(\gamma(\mathbf{R_b}) \wedge q_0 = 0)$ **return** *"no errors"*
endwhile
return $\mathbf{R_f}$

The $\gamma(\mathbf{R_f}) \rightarrow g$ implication check can be done efficiently through a multiple constrain image computation, without having to explicitly build the BDD for $\gamma(\mathbf{R_f})$.

Lemma 3 *Let* $\mathbf{R}$ *be an implicitly conjuncted list of BDDs* $(R_1, \ldots, R_p)$*, then* $\gamma(\mathbf{R}) \rightarrow g$ iff $Im(\gamma(\mathbf{R}), g) = \{1\}$.

Proof: *The proof relies on the observation that* $\gamma(\mathbf{R}) \rightarrow g$ *implies that the function* g *evaluates to* true *on every state in the set* $\gamma(\mathbf{R})$*. In other words, the image of the function* g *over the set* $\gamma(\mathbf{R})$ *can only be the singleton set* $\{1\}$.

Similarly, the $\gamma(\mathbf{R_b}) \wedge q_0 = 0$ check can be done efficiently through a multiple constrain image computation, without having to explicitly build the BDD for $\gamma(\mathbf{R_b})$.

Lemma 4 *Let* $\mathbf{R}$ *be an implicitly conjuncted list of BDDs* $(R_1, \ldots, R_p)$*, then* $(\gamma(\mathbf{R}) \wedge q_0) = 0$ iff $Im(\gamma(\mathbf{R}), q_0) = \{0\}$.

Proof: *The proof relies on the observation that* $(\gamma(\mathbf{R}) \wedge q_0) = 0$ *implies that the function* q_0 *evaluates to* false *on every state in the set* $\gamma(\mathbf{R})$. *In other words, the image of the function* q_0 *over the set* $\gamma(\mathbf{R})$ *can only be the singleton set* $\{0\}$.

If the function *BackAndForth* above is unable to prove the designer provided invariant g, then it returns a tuple $\mathbf{R}$, where $\gamma(\mathbf{R})$ represents the set of states that can be approximately reached from the initial set q_0, and which can approximately reach the error states $\neg g$. In order to further refine the over-approximation, we now choose a collection of larger subsets w, and repeat the whole process. However, while doing the forward and backward traversals this time around, we can still use the set $\mathbf{R}$ returned earlier, to constrain the search for a path to the error states.

5.5 Optimizations

The Pre_{dc} algorithm splits on the variables in w_j while computing the BDD for S_j. A simple optimization is to sort the order in which the splitting variables from w_j are picked. If $v_1 < v_2$ in the global BDD variable order, and $(v_1 \in w_j), (v_2 \in w_j)$, then we should split on v_1 before splitting on v_2 while computing S_j. This ensures that the BDD, being recursively created by Pre_{dc}, is aligned with the variable order at all times.

5.6 Counterexamples

If *BackAndForth* fails to rule out an error, it is useful to check whether there is an actual error by generating an example path from q_0 to a state that does not satisfy g. This both confirms the existence of an error and provides debugging information to the user. In exact reachability analysis, if an error state is reachable from an initial state, it is straightforward to construct a specific path from the initial state to an error. However, in this approximate analysis, such a path may or may not exist.

Starting from the error states, the algorithm computes approximate pre-images (intersected with the set of states seen in the previous pass) and stores the pre-images obtained at the various iterations of the fix-point algorithm in a stack. Let

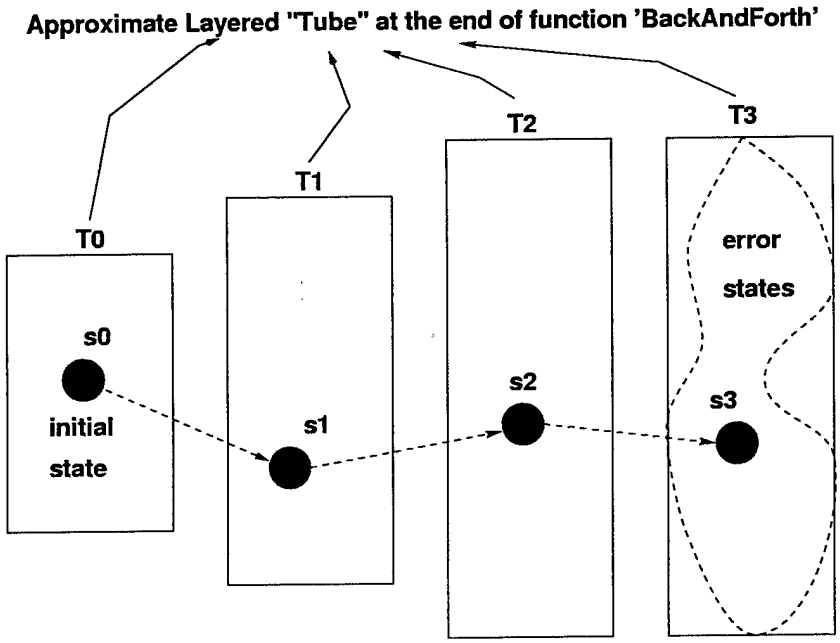


Figure 5.1: Counterexample generation from approximations

$T_0, T_1, \ldots, T_m$ (where T_m intersects with the error states, and T_0 intersects the initial states) be the final contents of the stack. The contents of the stack represent an approximate *tube* through which all counterexamples pass. Hence, the search for a counterexample can be restricted within the tube. (Note that each element of the stack is a tuple of BDDs whose concretization represents the states in each *layer* of the tube through which potential counterexamples must pass).

A *single* state, s_0, is chosen from the intersection $q_0 \wedge T_0$, and the exact image of s_0 is computed. If the image of s_0 intersects with T_1, a single state s_1 is chosen from the intersection and the process is repeated. (Figure 5.1 shows how the algorithm works). This simple heuristic was able to generate counterexamples over the design examples used here. This was partly because we assumed an overly general non-deterministic environment for the small design units, which made it easier to find some input assignment to lead us to the next layer.

However, note that this simple heuristic is not always guaranteed to succeed and may get *stuck* in some state s_j in layer T_j (where *stuck* in s_j means the image of s_j does not intersect with T_{j+1} at all, implying T_{j+1} is approximately reachable from s_j

but not exactly reachable from s_j). In Chapter 7, we will further improve on this simple heuristic to tackle this problem.

5.7 Experimental Results

The method was evaluated on a collection of control circuits from the MAGIC chip, a custom node controller in the Stanford FLASH multiprocessor [45]. The circuits are control intensive; the state bits do not include data path bits. Table 5.1 gives a brief description of the various control modules in the I/O unit.

Table 5.1: Control modules in I/O unit in FLASH

Module	State Bits	Input Bits
IOInboxQCtl	23	8
ReqDecode	37	27
ReqService	41	58
IOMiscBusCtl	44	18
PciInterface	88	55

The experimental implementation of the method was in LISP, calling David Long's BDD package (implemented in C) via the foreign function interface. The properties to prove were invariants provided by the designer. (Traditional benchmarks, such as ISCAS 89, do not come with specified properties, so they could not be used here.) The maximum number of BDD nodes was limited to 10 million nodes for each experiment. The variable subsets in $\mathbf{w} = (w_1, \ldots, w_p)$ were chosen manually using the heuristics described in Section 4.5.

Results

In the tables below, *Inv* lists the property to be proved. The column under P gives the results of the verification effort. A 'Y' means that property was proved, 'N' means a counterexample was generated, and '?' means that the verification exercise could not be completed.

The column labeled *Nodes* reports the maximum number of BDD nodes that existed at a time during the experiment, and *Time* reports the cpu time (in seconds) to complete the experiment on a MIPS R4300 with 768MB main memory (the cpu time includes time spent during LISP garbage collection).

The *Exact* column shows results of the exact pre-images of the error states, when it was possible to compute them. The exact pre-images were computed relative to the approximate reachable set computed during the first forward pass of the approximate algorithm. The same variable ordering was used in all the examples to get the numbers for the *Exact* method and the *Approximate* method.

Table 5.2: Note that our approximate scheme is able to prove every invariant that could be proved by the exact approach. Furthermore, in the case of properties $p3$ and $p5$, there is a decrease in the number of BDD nodes used by our approximate scheme, compared to the number of nodes used by the exact method.

Table 5.2: Proving IOInboxQCtl invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	4,216	9.5	Y	4,196	10.8
p2	Y	4,408	9.5	Y	4,312	10.8
p3	N	112,257	80.6	N	75,600	88.0
p4	Y	5,519	9.5	Y	4,850	10.8
p5	N	119,710	81.0	N	79,619	86.4

Table 5.3: These smaller examples (IOInboxQCtl and ReqDecode) demonstrate that our approximate scheme uses significantly fewer BDD nodes to prove the invariant, compared to the exact method. In the case of property $p2$, the difference is almost one order of magnitude.

Table 5.3: Proving ReqDecode invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	97,362	52.2	Y	42,954	49.9
p2	Y	680,107	76.1	Y	88,213	59.5

Table 5.4: Note that our approximate scheme is able to prove every invariant that

could be proved by the exact approach. As expected, the approximate approach takes fewer BDD nodes to prove the invariant.

Table 5.4: Proving ReqService invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	Y	95,598	517.8	Y	74,419	419.1
p2	N	121,573	1276.7	N	93,799	860.4
p3	Y	94,510	820.0	Y	74,419	418.5
p4	Y	112,367	1021.6	Y	94,365	418.5

Table 5.5: Note that the approximate scheme requires fewer BDD nodes to complete the verification exercise. The difference in the required number of nodes is also very large.

Table 5.5: Proving IOMiscBusCtl invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	N	2,936,929	1031.5	N	512,469	301.5
p2	Y	1,791,385	850.4	Y	426,324	302.3

Table 5.6: The approximate scheme is able to prove or disprove the property in all cases, unlike the exact method which fails to complete the verification exercise for most of the properties in the *PciInterface* design example. Furthermore, the approximate approach uses fewer BDD nodes to prove or disprove the invariant. The difference in the required number of BDD nodes is fairly large in most of the cases. Note that in the case of the *PciInterface* design example, the approximate method completes the verification exercise well within the 10 million node limit.

For the smaller example of *IOInboxQCtl*, the approximate method marginally takes more time than the exact method. The time advantage of the approximate method becomes clearer as we go for the larger design examples. Most of the time was spent in the approximate forward traversal (which was done for both the *Exact* and *Approximate* case).

The input environment for these design examples was assumed to be totally non-deterministic. The "errors" reported here were all because of such an overly general

Table 5.6: Proving PciInterface invariants

Inv	Exact			Approximate		
	P	Nodes	Time	P	Nodes	Time
p1	?	>10 mil	?	Y	1,012,742	559.8
p2	Y	1,116,686	2271.2	Y	1,007,843	661.6
p3	?	>10 mil	?	Y	1,324,916	750.9
p4	?	>10 mil	?	N	2,060,485	1290.6
p5	?	>10 mil	?	N	1,268,233	686.9
p6	?	>10 mil	?	N	2,097,440	973.6
p7	Y	1,113,254	468.8	Y	1,007,408	420.1

environment model. In Chapter 7 we will see an extension to this work after we have incorporated better environment models.

5.8 Conclusions

In this chapter, we have extended the idea of approximations using overlapping projections to symbolic *backward* reachability. We have combined it with a previous method of computing approximate *forward* reachable state sets. We show that approximate forward and backward reachability can be used in tandem to obtain more refined approximations. Overlapping projections are a viable approximation scheme and have helped to prove a number of designer provided invariants in a large design, where conventional exact approaches are rendered useless.

We have also proposed a new method to efficiently compute a sufficiently accurate pre-image during symbolic backward propagation using overlapping projections. Our method of domain splitting along with a number of associated optimizations has proved effective in tackling real, large design examples.

5.9 Appendix

5.9.1 Deterministic Relations

Recall from Section 2.3 that the underlying model for our applications is deterministic Mealy machines. The transition relation for such deterministic systems has some special properties that allows for distributing the pre-image over a conjunction. Before we delve further, it will help to define when a relation is *deterministic.*

Definition 4 *A relation* $T(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ *over* $[\mathbf{x} \rightarrow \mathcal{B}] \times [\mathbf{y} \rightarrow \mathcal{B}] \times [\mathbf{x}' \rightarrow \mathcal{B}]$ *is* single-valued *in* $(\mathbf{x}, \mathbf{y})$ *if for any assignment* $\mathbf{x}_0, \mathbf{y}_0$ *to* $\mathbf{x}, \mathbf{y}$, *there is a* unique *assignment* $\mathbf{x}'_0$ *to* $\mathbf{x}'$, *such that* $(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_0) \in T$. *Furthermore, the relation* $T(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ *is* total *over* $(\mathbf{x}, \mathbf{y})$ *if for any assignment* $\mathbf{x}_0, \mathbf{y}_0$ *to* $\mathbf{x}, \mathbf{y}$, *there is* at least *one assignment* $\mathbf{x}'_0$ *to* $\mathbf{x}'$, *such that* $(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_0) \in T$. *The relation* $T(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ *is* deterministic *over* $(\mathbf{x}, \mathbf{y})$ *if it is both single-valued and total over* $(\mathbf{x}, \mathbf{y})$.

Since we are dealing with *deterministic* synchronous hardware, for a given evaluation of the present state variables and the input bits, there is always a unique next state. Hence, the transition relation, $T(\mathbf{x}, \mathbf{y}, \mathbf{x}') = \wedge_{j=1}^{k}(x'_j \equiv n_j(\mathbf{x}, \mathbf{y}))$ in our Mealy machine model, is single-valued over $(\mathbf{x}, \mathbf{y})$. The *totality* assumption is not very restrictive, because even if a design is not *total* it can be made so by including an extra dummy state.

Pre-images sometimes distribute over conjunctions

Since we are dealing with implicit conjunctions in this thesis, a problem frequently encountered is computing the pre-image of a set $\mathbf{R}$, represented by an implicit conjunction, $\mathbf{R} = (R_1, \ldots, R_p)$. Since computing the BDD for $\gamma(\mathbf{R})$ through explicit conjunction is very likely to blow up, schemes which rely on building the BDD for $\gamma(\mathbf{R})$ and then computing its pre-image are unacceptable. Instead, it would be nice if we could compute the pre-images of the individual R_i's in $\mathbf{R}$ and then conjoin them. Unfortunately, the pre-image of a relation does *not* distribute over conjunctions. However,

under certain special conditions, the pre-image of an implicit conjunction $(R_1, \ldots, R_p)$ is the conjunction of the pre-images of the individual $R_i's$. Theorem 6 formally states the conditions.

Theorem 6 *Given a deterministic relation $T(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ over $(\mathbf{x}, \mathbf{y})$ and an implicit conjunction of BDDs $\mathbf{R} = (R_1, \ldots, R_p)$, then*

$$\begin{aligned}
&\exists \mathbf{y}, \mathbf{x}' \cdot [T(\mathbf{x}, \mathbf{y}, \mathbf{x}') \wedge R_1(\mathbf{x}') \ldots \wedge R_p(\mathbf{x}')] \\
&\qquad \equiv \\
&\exists \mathbf{y} \cdot [(\exists \mathbf{x}' \cdot [T(\mathbf{x}, \mathbf{y}, \mathbf{x}') \wedge R_1(\mathbf{x}')]) \wedge \\
&\qquad (\exists \mathbf{x}' \cdot [T(\mathbf{x}, \mathbf{y}, \mathbf{x}') \wedge R_2(\mathbf{x}')]) \wedge \\
&\qquad \ldots \\
&\qquad (\exists \mathbf{x}' \cdot [T(\mathbf{x}, \mathbf{y}, \mathbf{x}') \wedge R_p(\mathbf{x}')])].
\end{aligned}$$

Proof: *The equivalence can be proved by proving implication in both directions. Proof for LHS $\rightarrow$ RHS is trivial. We give here a proof for the other direction, RHS $\rightarrow$ LHS. Let us consider a point $\mathbf{x}_0 \in$ RHS. Hence there is some assignment y_0 to $\mathbf{y}$ and some assignment $\mathbf{x}'_1, \ldots, \mathbf{x}'_p$ to the various instances of $\mathbf{x}'$ in the RHS, such that*

$$\begin{aligned}
&((T(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_1) \wedge R_1(\mathbf{x}'_1)) \quad \wedge \\
&(T(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_2) \wedge R_2(\mathbf{x}'_2)) \quad \wedge \\
&\qquad \ldots \\
&(T(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_p) \wedge R_p(\mathbf{x}'_p)))
\end{aligned}$$

is true. However since the relation $T(\mathbf{x}, \mathbf{y}, \mathbf{x}')$ is deterministic over $(\mathbf{x}, \mathbf{y})$; for a given assignment $(\mathbf{x}_0, \mathbf{y}_0)$ to $(\mathbf{x}, \mathbf{y})$, there can be one and only one assignment to $\mathbf{x}'$ that makes T true. Hence $\mathbf{x}'_1 = \mathbf{x}'_2 = \ldots = \mathbf{x}'_p$. Hence the following must be true

$$(T(\mathbf{x}_0, \mathbf{y}_0, \mathbf{x}'_1) \wedge R_1(\mathbf{x}'_1) \wedge R_2(\mathbf{x}'_1) \wedge \ldots \wedge R_p(\mathbf{x}'_1))$$

This implies $\mathbf{x}_0 \in$ *LHS (with* $(\mathbf{y}_0, \mathbf{x}'_1)$ *serving as the witness), which completes the proof.*

5.9.2 Satisfiability Check with Multiple Constrain

After the BDD for a function is built, checking for satisfiability is easy. This is because BDDs are canonical for a given variable ordering. However, when dealing with implicit conjunctions of BDDs, the canonicity property is lost. The same set of states can be represented by more than one implicitly conjoined list of BDDs. This makes the problem of checking for satisfiability of implicitly conjoined lists non-trivial.

Lemma 5 *Let* $\mathbf{R} = (R_1, \ldots, R_p)$ *be an implicitly conjoined list of BDDs, then* $\gamma(\mathbf{R}) = 0$ iff $Im(\bigwedge_{i=2}^{p} R_i, R_1) = \{0\}$.

Proof: The proof relies on the observation that $\gamma(\mathbf{R}) = 0$ implies that $R_1 \wedge (\bigwedge_{i=2}^{p} R_i) = 0$. This implies that the function R_1 evaluates to *false* on every state in the set $\bigwedge_{i=2}^{p} R_i$. In other words, the image of the function R_1 over the set $\bigwedge_{i=2}^{p} R_i$ can only be the singleton set $\{0\}$. Note that the multiple constrain operator can be used to compute this image without doing the explicit conjunction. In fact, the Lemma above can be generalized to pick any element (and not just R_1) of the tuple $\mathbf{R} = (R_1, \ldots, R_p)$ and compute its image over the conjunction of the other elements of the tuple. Let R_j be an element of the tuple $\mathbf{R}$, then $\gamma(\mathbf{R}) = 0$ *iff* $Im(\bigwedge_{(i=1)(i\neq j)}^{p} R_i, R_j) = \{0\}$.

Chapter 6

Auxiliary State Variables

The schemes discussed thus far in this thesis rely on doing approximate reachability over overlapping subsets of the state variables. These schemes can be further improved upon by augmenting the set of state variables with some *auxiliary state variables.* This chapter starts with the definition and intuition behind auxiliary state variables. Thereafter, the technical challenges involved in creating auxiliary state variables is elaborated upon. Finally, the results obtained by applying this method to different design examples are presented.

6.1 Using Internal Abstractions

In this thesis, we have modeled synchronous digital designs with a Mealy machine model, where the logic between register boundaries is flattened and a next state function is assigned to each state holding element. Sometimes, wires hidden deep inside the combinational logic carry a lot of useful information that can help capture the communication and correlation between state machines. Unfortunately, there is no state variable that explicitly captures the information embedded inside some of these internal wires.

In order to exploit these internal abstractions hidden inside the combinational logic, the design needs to be augmented with some special state variables that can capture the information inside these wires, but which at the same time do not change

the externally visible behavior of the design. This is achieved through auxiliary state variables. An auxiliary variable is an internal state component that is added to the implementation without affecting the externally visible behavior.

6.1.1 Key Intuition

The key observation which makes auxiliary state variables useful in our applications is that different state machines in a design often have a *narrow communication interface*; in other words, the number of bits of information communicated between state machines is usually small, even though the number of bits needed to encode the state of these machines is relatively large. If there are no explicit state variables that capture the information being communicated between these machines, there is no option but to use the bits encoding the states of the two machines. And since the communication width is often much less than the number of bits in the state of each machine, this often leads to unnecessarily large subsets that can potentially suffer from BDD blowup problems. The following example brings out this point.

6.1.2 Example to Illustrate Power of Auxiliary Variables

Consider the simple design shown in Figure 6.1. The design has 96 state variables, denoted by $(x_1, \ldots, x_{96})$. The *Equality Detector* checks whether the 32 bit state vector of the two state machines (FSM_1 and FSM_2) is identical, and then passes its output to all the state machines. Exact reachability would require computing images over the variables $(x_1, \ldots, x_{96})$. Intermediate image BDDs with such large support sets often blow up. Alternatively, we could choose to do approximate reachability over the disjoint [12] subsets $(x_1, \ldots, x_{32})$, $(x_{33}, \ldots, x_{64})$ and $(x_{64}, \ldots, x_{96})$. Since the subsets have 32 variables, the intermediate image BDDs have 32 variables in their support and are less likely to blow up. However, there is a price to the loss of accuracy, since interaction between the variables in different subsets is lost. Using overlapping projections [29], we could capture some interaction by choosing the subsets $(x_1, \ldots, x_{64})$, $(x_{33}, \ldots, x_{96})$ and $(x_1, \ldots, x_{32}, x_{65}, \ldots, x_{96})$. The intermediate image BDDs have 64 variables in their support, but it captures more interaction between the state variables

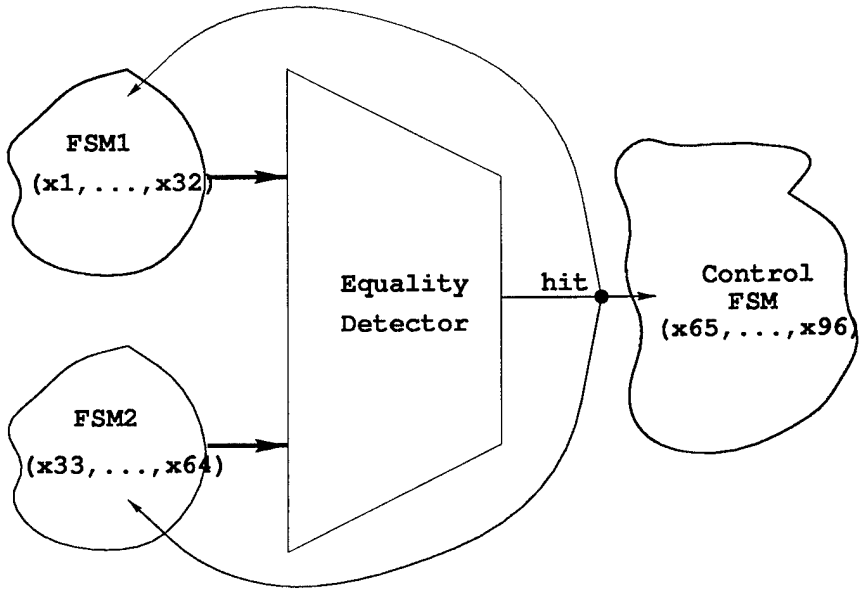


Figure 6.1: Example to illustrate potential of using auxiliary variables

than the disjoint partition case.

However, the only interaction between state variables $(x_{65}, \ldots, x_{96})$ and the other state variables happens through the signal *hit*. Hence, there is a single bit of information being communicated between the state machines. Unfortunately, there is no single state variable that captures this bit of information hidden deep inside the combinational logic. It appears wasteful to add 64 state variables, $(x_1, \ldots, x_{64})$, to other subsets in order to transmit one bit of information to other subsets. Instead, we propose introducing an auxiliary state variable for the wire *hit*. Now interaction between the state variables is captured by choosing the subsets $(x_1, \ldots, x_{32}, hit)$, $(x_{33}, \ldots, x_{64}, hit)$, $(x_{65}, \ldots, x_{96}, hit)$, and doing symbolic reachability [29] over them. The largest subset in this case is size 33, but it captures the critical correlation between all 96 state variables in the design.

The benefit of looking for important internal conditions in the combinational logic, representing *narrow communication interfaces* between state machines, and converting them to auxiliary variables is now clear: an auxiliary variable captures important properties of many state variables into a *single* new state bit. This can be added to the other subsets to capture the correlation between many state variables,

even as the number of variables in different subsets is small.

6.1.3 Related Work

Augmenting a legal implementation with some extra state components in a way that places no constraints on the behavior of the implementation is not an entirely new idea. Abadi and Lamport [1] introduced a special class of auxiliary variables, *history* and *prophecy* variables, to broaden the applicability of refinement mapping techniques. We use auxiliary state variables [31] to broaden applicability of approximate reachability techniques. Note that this contrasts to the idea of extracting functional dependencies [40, 67] and removing extra state variables to simplify the model of the underlying design.

6.2 Converting Internal Wires to Auxiliary State Variable

Before we treat auxiliary variables as first class state variables, we need to assign a next state function and an initial state to them. They can then be incorporated into our Mealy machine model and then the algorithms from the preceding chapters can easily be applied.

6.2.1 Next State Function for Auxiliary Variables

In order to illustrate how we assign a next state function to auxiliary variables, we start with a typical design, as shown in Figure 6.2. It has a set of state holding elements ($\mathbf{x} = (x_1, x_2, x_3)$ in Figure 6.2) and some combinational logic. Each state variable has an associated next state function logic ((n_1, n_2, n_3) in Figure 6.2). Let a be some internal wire in the design, and let $a = g(\mathbf{x})$ be the function that determines the value of a in time t as a function of the state variables $\mathbf{x}$ at time t.

If we let the subscript denote the time stamp, we have: $a_t = g(\mathbf{x}_t)$ and $a_{t+1} = g(\mathbf{x}_{t+1})$. Using $\mathbf{x}_{t+1} = \mathbf{n}(\mathbf{x}_t, \mathbf{y}_t)$, we get $a_{t+1} = g(\mathbf{n}(\mathbf{x}_t, \mathbf{y}_t))$, which is the required next

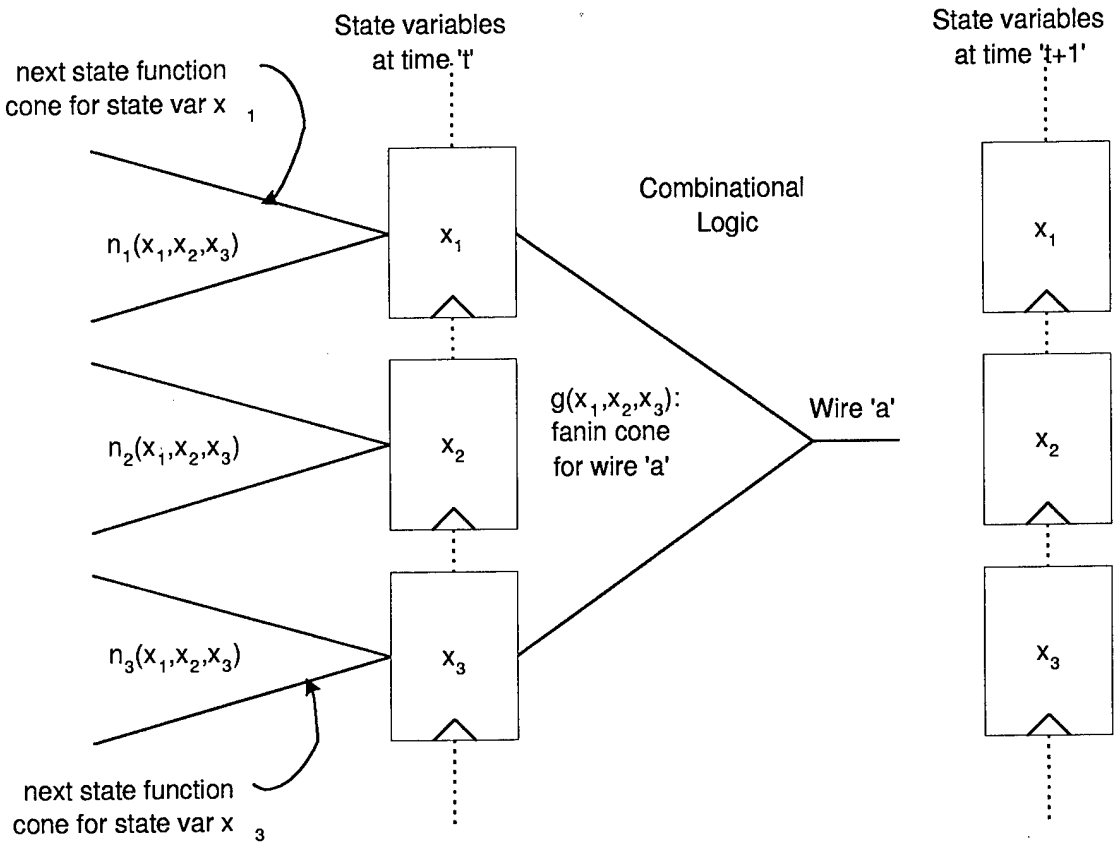


Figure 6.2: Typical design

state function for auxiliary state variable a.

This transformation is shown in Figure 6.3. For the example in Figure 6.1, let $g(x_1, \ldots, x_{64})$ be the Boolean function for the cone of logic feeding into the wire *hit*. Furthermore, let $(n_1, \ldots, n_{64})$ be the next state functions for the usual state variables $(x_1, \ldots, x_{64})$. The next state function for auxiliary state variable *hit* is obtained by substituting n_i for x_i in $g(x_1, \ldots, x_{64})$. This has the effect of retiming the internal wire *hit*.

Note that we would not have been able to do the transformation above if g involved some input variables in its support. If $a = g(\mathbf{x}, \mathbf{y})$ (where $\mathbf{y}$ is the input bits) then $a_{t+1} = g(\mathbf{x}_{t+1}, \mathbf{y}_{t+1})$ and we cannot represent the inputs in the next cycle, $\mathbf{y}_{t+1}$, in terms of $\mathbf{x}_t$ and $\mathbf{y}_t$. This limitation can be circumvented by including the inputs as part of the state. We never used the following for any of our results here, but if we want to convert internal wires that also have inputs in their fanin cone into auxiliary variables, the Mealy machine $M = \langle \mathbf{x}, \mathbf{y}, q_0, \mathbf{n} \rangle$, can be transformed to another Mealy machine $M' = \langle \mathbf{x}', \mathbf{y}', q_0', \mathbf{n}' \rangle$, where $\mathbf{x}' = \mathbf{x} \cup \mathbf{y}$ and the initial condition q_0' is set to q_0. The $\mathbf{y}'$ component is a set with a primed version for each variable in $\mathbf{y}$. The next state function for the $\mathbf{x}$ state variables remains the same, but for the $\mathbf{y}$ variables, their next state function is the corresponding input variable from $\mathbf{y}'$. Assuming a totally unconstrained input environment, the machines M and M' allow the same externally visible behaviors and hence have the same set of reachable states (projected on to the $\mathbf{x}$ variables). However, M' allows more flexibility in choosing auxiliary state variables.

6.2.2 Initial Condition for Auxiliary Variables

The auxiliary state variables need to be initialized. Let $\mathbf{a} : (a_1, \ldots, a_m)$ be the list of auxiliary variables and $\mathbf{g} : (g_1, \ldots, g_m)$ be the list of Boolean functions (represented as BDDs) such that $g_i(\mathbf{x})$ determines the value of a_i at time t in terms of state variables $\mathbf{x}$ at time t. The initial condition for the $\mathbf{a} : (a_1, \ldots, a_m)$ variables is obtained by the following image computation, $Im(q_0, \mathbf{g})$. In our applications, initial condition q_0 is a single state, and this reduces the image computation problem to computing $g_i(\mathbf{x}) \downarrow q_0$ for each auxiliary variable a_i. (The $\downarrow$ is the generalized cofactor [18] operator).

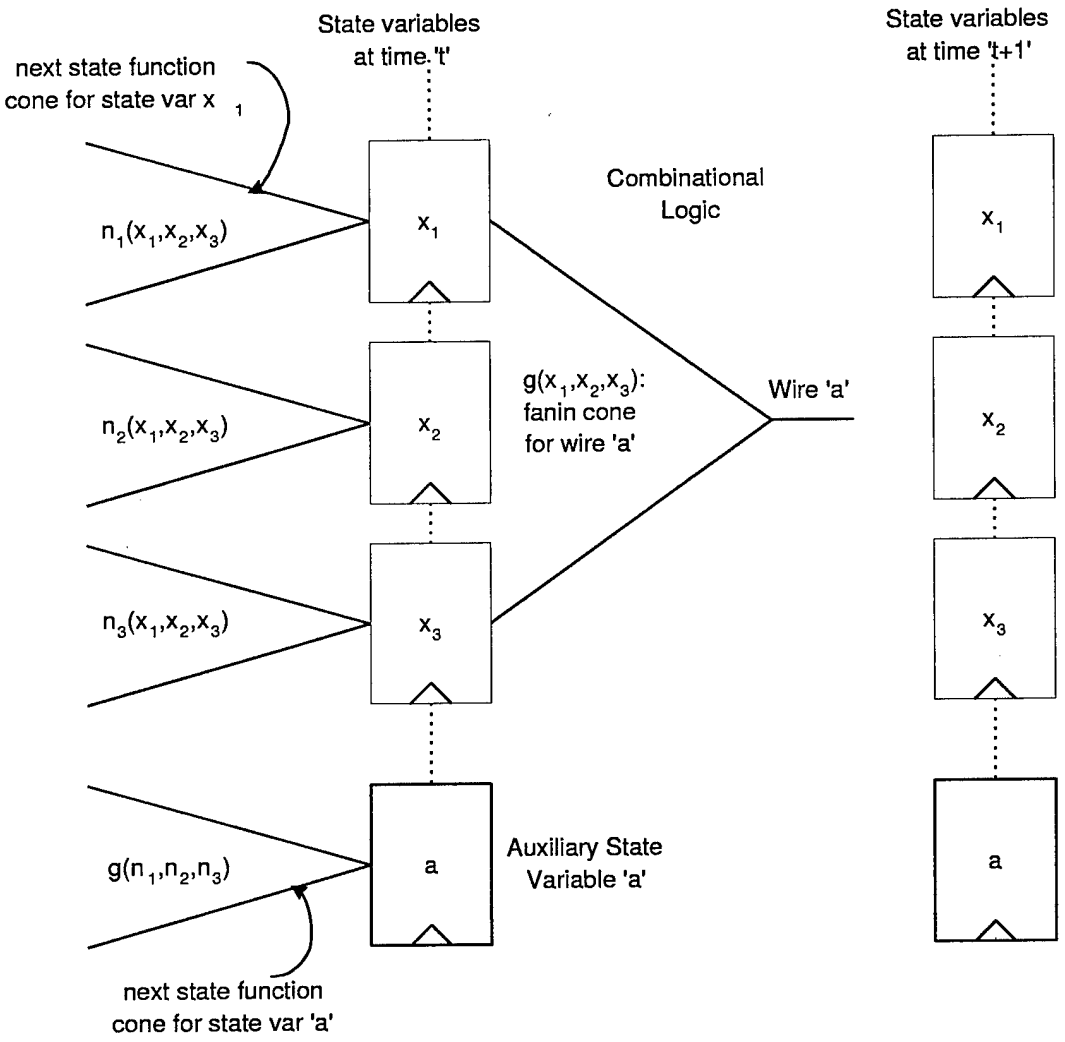


Figure 6.3: Design including auxiliary state variables

6.3 Heuristics to Choose Auxiliary State Variables

The scheme for choosing which internal abstractions to convert to auxiliary state variables is presently manual, and relies on being able to inspect the RTL source. It helps to look at the RTL source, because designers often create internal abstractions themselves, while coding up their design using a hardware description language (such as Verilog). Hence, we can leverage off this high level information directly by inspecting the RTL description.

First, the FSMs are identified by inspecting the Verilog source. The next state transition for every FSM was typically encoded as part of an *always* block in the Verilog source. By inspecting the *always* block, it is possible to extract the internal wires that affect the next state transition of each FSM. In turn, if those internal wires depend only on state variables, they are chosen as auxiliary state variables.

However, the gate level descriptions of circuits like the ISCAS 89 benchmark circuits, are devoid of any high level information. For such circuits, internal wires which have a *high fanin* and *high fanout*, and are at the same time solely determined by the state variables in the design (*i.e.*, their fanin cones involve only state variables), are identified. The intuition behind this heuristic is that such high fanin internal wires carry some information about the large number of state variables in their fanin cone. Furthermore, since they have high fanout, they transmit this information to a large number of other state variables. Hence, including these wires as auxiliary state variables in other subsets of $\mathbf{w}$ captures some correlation between the state variables in the other subsets and the large number of state variables in the fanin cone of the internal wire.

6.4 Experimental Results

The experimental implementation of the method was in LISP, calling David Long's BDD package (implemented in C) via the foreign function interface. The method was evaluated on a collection of control circuits from the MAGIC chip, a custom node controller in the Stanford FLASH multiprocessor [45]. For comparison with

earlier work, results obtained by applying the idea to the publicly available ISCAS89 benchmark circuits are also presented. The approximate algorithm returns a superset of the reachable states, which is also an invariant of the design. To quantify the size of the superset, the satisfying fraction of the the superset is computed (please refer to the Appendix of this chapter in Section 6.6, for the algorithm that was used to compute an upper bound on the satisfying fraction). Since projection induces an over-approximation, the smaller the satisfying fraction, the stronger the invariant.

6.4.1 Results on Design Examples from FLASH

Table 6.1 gives a brief description of the sizes of various control modules extracted from the I/O unit in terms of the number of state variables, auxiliary state variables and input variables. (IOQ_ReqD stands for the module obtained by combining the submodules IOInboxQCtl and ReqDecode, whereas ReqS_ReqD stands for the module obtained by combining ReqService and ReqDecode).

Table 6.1: Control modules in I/O unit in FLASH

Module	State	Auxiliary	Total	Input
IOQ_ReqD	60	6	66	25
ReqS_ReqD	78	14	92	48
PciInterface	88	20	108	55

Table 6.2: IOQ_ReqD: Size of approx. reachable set with auxiliary variables

Subsets	Usual State Variables			Adding Auxiliary Variables			Ratio
	Sat. Fr.	Time	Nodes	Sat. Fr.	Time	Nodes	
$\mathbf{w}_1$	2.570e-08	22	63,180	1.485e-09	48	97,685	1.731e+01
$\mathbf{w}_2$	"	"	"	1.399e-09	60	111,517	1.838e+01

The maximum number of BDD nodes (BDD Node Limit) for each experiment (*i.e.* for each row in the following tables) was preset. Initially, the collection of subsets, $\mathbf{w}$, has small-sized, possibly-overlapping subsets over the usual state variables alone. These subsets incrementally become larger, until the experiment requires more BDD

Table 6.3: ReqS_ReqD: Size of approx. reachable set with auxiliary variables

Subsets	Usual State Variables			Adding Auxiliary Variables			Ratio
	Sat. Fr.	Time	Nodes	Sat. Fr.	Time	Nodes	
$\mathbf{w}_1$	3.835e-09	553	644,667	2.632e-09	1,302	846,476	1.457
$\mathbf{w}_2$	"	"	"	2.282e-09	1,232	1,832,354	1.680

nodes than set in the limit. To this collection of subsets giving the best result within the node limit, extra auxiliary bits were added as per the heuristics given earlier (Section 6.3). Thus, by staying within the bounds of the node limit, the strongest invariant obtained with overlapping projections over usual state variables is compared to the strongest invariant obtained with overlapping projections over the augmented (usual and auxiliary) set of state variables. The column *Subsets* lists different choice of the collection of subsets, $\mathbf{w}$, where the size of subsets increases as we go down a table. The same variable ordering was used for both the schemes.

The column labeled *Nodes* keeps track of the highest number of nodes that existed at a time during the experiment. The *Time* column lists the cpu time (in seconds) to complete the experiment on a MIPS R4300 with 768MB of memory (the cpu time includes the time spent doing LISP garbage collection). The last column under the heading *Ratio* is the ratio between the satisfying fraction obtained by using usual state variables alone and the satisfying fraction obtained on adding auxiliary variables. Thus, larger figures in the *Ratio* column indicate better results with auxiliary variables. Note the order of magnitude improvement reported in the *IOQ_ReqD* example.

Table 6.4: PciInterface: Size of approx. reachable set with auxiliary variables

Subsets	Usual State Variables			Adding Auxiliary Variables			Ratio
	Sat. Fr.	Time	Nodes	Sat. Fr.	Time	Nodes	
$\mathbf{w}_1$	1.801e-05	308	466441	5.892e-06	1,471	971,880	3.057
$\mathbf{w}_2$	2.175e-06	2,907	1,260,260	7.003e-07	9,174	8,349,050	3.105

6.4.2 Results on ISCAS-89 Benchmark Circuits

The algorithm was also evaluated on the bigger benchmarks in ISCAS 89 benchmark suite. Once again, the partitions used by Cho *et al.* [12] were used to identify the FSMs in the design. To these partitions, small overlaps were added to report the numbers in Table 4.7 to show the potential of approximate reachability on overlapping subsets of the usual state variables. Some auxiliary state variables are added to some of the overlapping subsets, and results are compared with those in Table 4.7. Table 6.5 gives a brief description of the sizes and number of auxiliary variables added to the various benchmark circuits. Table 6.6 has the details on the improvement achieved by using auxiliary state variables. The *Iter* column lists the number of iterations needed to reach the fix-point.

The new algorithm was also tried on circuit s1423, but unfortunately we could not improve upon the results reported in Table 4.7. (We suspect it is because s1423 has a highly interconnected state transition graph (STG). Some high level insight into the design, which ISCAS benchmark circuits lack, could better guide the choice of auxiliary variables). However, for s13207, s15850 and s38584, an improvement by at least an order of magnitude is reported.

Table 6.5: Auxiliary variables added to ISCAS 89 circuits

Circuit	State	Auxiliary	Total	Input
s13207	669	39	708	31
s15850	597	14	611	14
s38584	1452	12	1464	12

Table 6.6: ISCAS 89 circuits: Size of approximate reachable set with auxiliary variables

Circuit	Usual State Variables			Adding Auxiliary Variables			Ratio
	Sat. Fr.	Iter	Nodes	Sat. Fr.	Iter	Nodes	
s13207	1.14e-115	10+5	198,779	1.24e-117	10+5	1,171,473	9.2e+01
s15850	3.94e-102	10+4	336,048	3.92e-103	10+4	339,031	1.0e+01
s38584	5.76e-57	10+5	1,853,461	1.20e-58	10+4	1,952,730	4.8e+01

Given the large number of state variables in these circuits, and that the various subsets have overlapping support, it is very difficult to compute the size of the approximate reachable set. The numbers in Table 6.6 under the *Sat Fr* column for *Auxiliary Variables* are *upper bounds* on the size of the reachable set. (Please refer to the Appendix of this chapter in Section 6.6 for the algorithm used to compute an upper bound on the size of the approximate reachable set). The true size of the approximate reachable set using auxiliary state variables is much smaller than what is reported here.

Note that the TMBM algorithm [12] was used for these benchmarks. TMBM starts off as TFBF [12] and then switches to MBM [12] after a few iterations. The *Iter* column in Table 6.6 lists the number of iterations of doing TFBF + the number of iterations in the outer greatest fix-point of MBM.

6.5 Conclusions

The key observation that makes auxiliary variables a good idea for this application is that the communication width or the number of bits of information communicated between state machines is often much lower than the number of bits encoding the states of these machines. Capturing the information in the interface between machines through special state variables enables the capturing of communication between state machines with smaller-sized subsets. The experiments show that a few appropriately chosen internal conditions added as auxiliary variables can substantially improve the quality of the over-approximation.

6.6 Appendix

6.6.1 *Sat_Fr* of Superset for FLASH I/O circuits

In Section 4.8, an algorithm to compute an upper bound on the satisfying fraction of an implicit conjunction of BDDs was presented. With auxiliary state variables the

problem needs to be slightly modified. Given a list of BDDs $\mathbf{S} : (S_1, \ldots, S_p)$, corresponding to the collection of possibly overlapping subsets $\mathbf{w} : (w_1, \ldots, w_p)$, compute *sat_fr* of $\gamma(\mathbf{S})$. The only difference now is that the individual w_j subsets in $\mathbf{w}$ may include auxiliary state variables, which can artificially lower the satisfying fraction, and hence gives a distorted picture of the number of states in the superset. For the design examples from the FLASH example, it was possible to remove the auxiliary variables from $\mathbf{S}$ and accurately compute the satisfying fraction. The details of the algorithm are given below.

Let $a : (a_1, \ldots, a_m)$ be the set of auxiliary state variables. Corresponding to each auxiliary state variable a_i, let $g_i(\mathbf{x})$ be the Boolean function (represented as a BDD) which determines the value of the auxiliary state variable a_i in time t as a function of the value of the usual state variables at time t. The algorithm defined below substitutes the function g_i for every instance of a_i in the elements of the list $\mathbf{S}$. At this point, $\mathbf{S}$ has only the usual state variables in its support. Then the algorithm explicitly computes $\gamma(\mathbf{S})$ and finds its satisfying fraction.

for *j=1* **up to** p **by** *1* **do**
  for *i=1* **up to** m **by** *1* **do**
    Substitute g_i for every instance of a_i in S_j
  endfor
endfor
Compute *final_bdd* = $\wedge_{j=1}^{p} S_j$
return *sat_fr (final_bdd)*

For the larger ISCAS 89 benchmark circuits, because of BDD size blowup problems, it was not feasible to remove all the auxiliary state variables and explicitly compute *final_bdd* = $\gamma(\mathbf{S})$. Hence, the conservative algorithm given in Section 4.8 was used and then normalized to compensate for the increase in the number of state variables. (If m is the number of auxiliary state variables added, the result obtained from the algorithm in Section 4.8 was multiplied by 2^m to obtain an upper bound on the satisfying fraction for the reachable states over the usual state variables alone).

The Monte Carlo simulation technique, an alternative method to estimate the satisfying fraction of $\gamma(\mathbf{S})$, appears to be ineffective because of the extreme sparseness of the state space covered by $\gamma(\mathbf{S})$. To get estimates with a good confidence interval, a prohibitively large number of samples would be needed.

Chapter 7

Counterexamples

Example is always more efficacious than precept. — Samuel Johnson.

The verification algorithms presented thus far in this thesis are susceptible to false negatives. Even though a property holds, the approximate algorithms may not be able to prove it to be so. Searching for real counterexamples in such an approximate space is liable to failure. In this chapter, the "hybridization effect" induced by our approximation scheme is identified as the cause for the failure. A heuristic based on Hamming Distance is proposed to improve the choice of projections that reduces the hybridization effect and facilitates either a genuine counterexample or proof of the property. Finally, the results obtained on a large real design example are presented.

7.1 Introduction

One of the key desirable features of model checkers is their ability to generate counterexamples automatically, which can directly aid the debugging of the design. However, approximate model checking techniques have the drawback that it is not always feasible to map a counterexample generated in the approximate space into a valid counterexample in the real design. The approximated design may have extra degrees of freedom, allowing certain transitions not possible in the real design. Analysis of

the cause of failure of the counterexample in the approximated space can highlight what information is lost in the approximation process, and then hints can be given that appropriately refine the approximation.

7.2 Related Work

Consider the following four-step general iterative approach to formal verification.

1. **Initial approximation:** Choose an initial approximation.

2. **Verification:** Try to verify the property. If the verification is successful, terminate with success. Otherwise, go to step 3.

3. **Failure Diagnosis:** Analyze the failure report from the verification algorithm and determine whether the failure is inherent in the original design or because of the approximation. If the former is true, terminate with failure. If the latter is true, go to step 4.

4. **Refinement:** Refine the approximation in a way that the reported bogus failure is eliminated. Go back to step 2.

This general approach is applicable to any formal verification technique that allows for conservative simplifications. The specific algorithm will depend on the technique, heuristic choices of initial abstraction, failure report and refinement procedures.

As expected, this basic idea has been used by various researchers. Kurshan [44] used it in the context of verification of timed automata, while Balarin *et al.* [3] used it to check for language containment. Clarke *et al.* [16] explored this same basic idea in verification using abstraction functions for different variables in a SMV program. We explore the same basic idea in the context of approximation by overlapping projections [29].

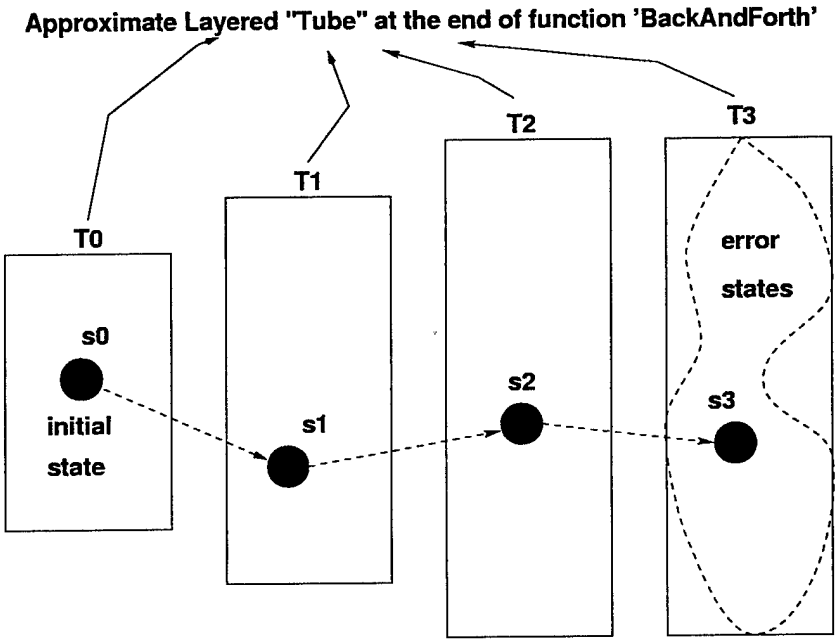


Figure 7.1: Counterexample generation from approximations

7.3 Hybridization

If the algorithm *BackAndForth* (Section 5.4) fails to rule out an error, it is useful to check whether there is an actual error by generating an example path from q_0 to a state that does not satisfy g. This both confirms the existence of an error and provides debugging information to the user. In Section 5.6, we saw a simple heuristic that searches for a counterexample in the approximate space returned by the algorithm *BackAndForth*.

Figure 7.1 is one way to visualize the idea behind the heuristic, where we try making one step transitions from a state in the present layer to some state in the next layer (details in Section 5.6). This simple heuristic is not guaranteed to succeed and can get stuck in some *bogus* state s_j in layer T_j, which means that paths from the initial states to these states in layer T_j cannot be extended to form a complete counterexample. It is useful to analyze what information is lost in the approximation scheme that allows such bogus states to creep into the *tube*. Then hints can be provided on how to improve the choice of projections and thereby create more accurate

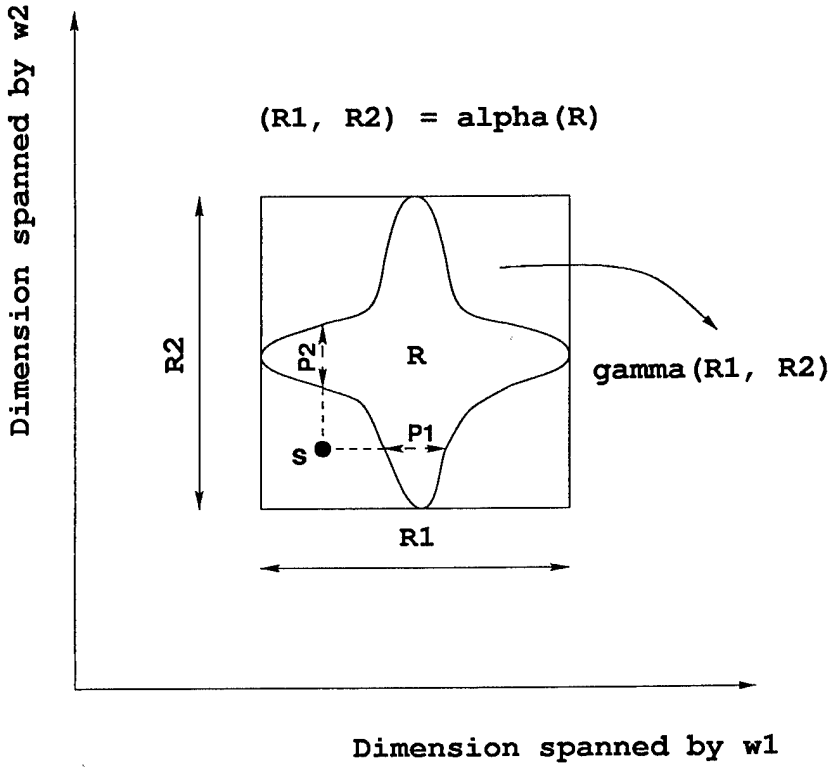


Figure 7.2: Hybridization effect induced by projections

tubes with fewer bogus states.

7.3.1 How do Bogus States Creep in?

In order to understand how bogus states creep into the approximation tube, we need to understand the approximation induced by projections. The geometric interpretation of the approximation induced is in Figure 7.2. For simplicity, assume there are only two subsets in our collection of subsets (the ideas presented can be extended to an arbitrary number of subsets). The irregular shape in Figure 7.2 represents the exact set R, and the outer box represents the set of states obtained after projecting through α, then concretizing through γ. This allows bogus states like s in Figure 7.2 to creep into the approximation tube. For the given choice of subsets, there is some loss of correlation between the state variables in different subsets. In particular, bogus states like s do not have to agree with some real state in R across all the w_1 and w_2 bits, but

instead merely need to agree with some real states in R on w_1 bits, and with some *other* real states in R on w_2 bits. This leads to the notion of *hybridization.*

Definition 5 *Let s_1 and s_2 be two states from $[\mathbf{x} \rightarrow \mathcal{B}]$. Given a collection of subsets $w = (w_1, w_2)$, the states s_1 and s_2 are said to* hybridize *a state s,* i.e., *$s \in hybrid(s_1, s_2)$ if the following conditions hold:*

- *$s \neq s_1$ and $s \neq s_2$, and*
- *$\alpha_1(s) = \alpha_1(s_1)$, and*
- *$\alpha_2(s) = \alpha_2(s_2)$*

In other words, $s \in hybrid(s_1, s_2)$ holds relative to the choice of subsets $w = (w_1, w_2)$ if s agrees with s_1 on the w_1 bits and s agrees with s_2 on the w_2 bits. From Figure 7.2, note that every state from P_1 would hybridize with every state from P_2 to allow bogus state s to creep in.

Example 6 *Let s_1 and s_2 be two states from P_1 and P_2, respectively. Since s_1, s_2 and s are single states, they have a unique assignment to all the state variables in x. For ease of exposition, consider a design with six state variables $x = \{a, b, c, d, e, f\}$, $R = b \vee e$, $w_1 = \{a, b, c, d\}$ and $w_2 = \{c, d, e, f\}$. Let the bit vectors $s = 101101$, $s_1 = 101111$ and $s_2 = 111101$ represent these single states (the leftmost bit in the vector refers to variable 'a' and the rightmost refers to variable 'f'). Note that $s_1 \in R$, $s_2 \in R$, but $s \notin R$ as required from Figure 7.2. Also s and s_1 differ in the assignment to variable e, whereas s and s_2 differ in the assignment to variable b.*

7.3.2 Intuition to Removing Bogus States

The key idea is that by looking at the bits that disagree in s and s_1 (s_2), and adding them to the w_1 (w_2) subset, the bogus state s can be eliminated from the approximation. Consider adding to w_1 the bit positions where s_1 and s differ, *i.e.*, $w_1' = w_1 \cup \{e\}$. The new $w_2' = w_2 \cup \{b\}$ is formed analogously by adding w_2 to the bit variables on which s_2 and s differ. Relative to this new choice of subsets $w' = (w_1', w_2')$, the bogus

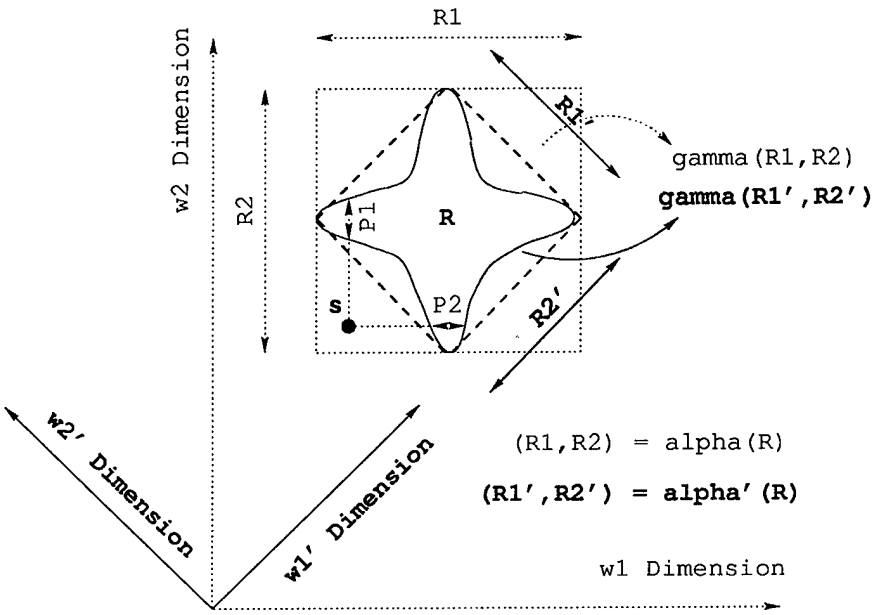


Figure 7.3: Refinement through Hamming distance heuristic

state s is no longer a hybrid state. This is because the possible states that could hybridize to give place to the state s, namely P_1' and P_2', are both reduced to 0. (Relative to this new choice of subsets, $P_1' = \alpha_1'(s) \wedge R$ reduces to 0. Similarly $P_2' = \alpha_2'(s) \wedge R$ also reduces to 0.)

The geometric interpretation of the refinement induced by this heuristic is in Figure 7.3. Relative to the earlier choice of subsets $w = (w_1, w_2)$, we use the pair of functions $\langle \alpha, \gamma \rangle$ and the set R can be approximated by the tuple (R_1, R_2). However, this includes the bogus state s since $s \in \gamma(R_1, R_2)$. With the new choice of subsets $w' = (w_1', w_2')$, we use the associated pair of functions $\langle \alpha', \gamma' \rangle$ and the set R is approximated by the tuple (R_1', R_2'). However, this does *not* allow the bogus state s to creep in, since $s \notin \gamma'(R_1', R_2')$. Also note that $\gamma'(R_1', R_2') \subseteq \gamma(R_1, R_2)$, implying that results from the choice of subsets $w' = (w_1', w_2')$ are guaranteed to be tighter approximations than those obtained with $w = (w_1, w_2)$.

However, as the approximations get refined, the size of the individual subsets in w' grows. Larger subsets yield more accurate results; however, they are more likely to suffer from BDD size blowup during the fix-point routines in *BackAndForth*. To ensure that the sizes of the individual subsets grow incrementally, it is advisable

to choose states s_1, s_2 from $P_1 and P_2$, respectively, such that they have the smallest *Hamming distance* [36] from s.[1] This will incrementally lead to one or more iterations of augmenting the subsets that will rule out the bogus state s. Formally, the algorithm for improving the choice of subsets is:

function *ImproveProj* $((P_1, P_2), s, (w_1, w_2))$
  Choose $s_1 \in P_1$ *s.t.* $\mid s_1 - s \mid$ *is small*
  Choose $s_2 \in P_2$ *s.t.* $\mid s_2 - s \mid$ *is small*
  $w_1' = w_1 \cup$ *bits where* (s, s_1) *differ*
  $w_2' = w_2 \cup$ *bits where* (s, s_2) *differ*
return $w' = (w_1', w_2')$

Choosing the states s_1 and s_2 in the algorithm above requires finding a state from a set of states that has minimum Hamming distance from some other reference state. An efficient algorithm proposed by Yang [68, 69] was used here. The complexity of that algorithm is linear in the size of the BDD representing the set. In the general case of more than two subsets in the collection, $w = (w_1, \ldots, w_p)$, the subset w_i is improved relative to the next subset in the collection, *i.e.* $w_{(i+1) \bmod p}$.

7.4 Hamming Distance Heuristic

It is useful to distinguish between two different kinds of bogus states. Suppose the counterexample generation method is stuck at a state s in some layer (R_1, R_2). Depending on whether or not the approximate image of s intersects with the next layer, there are two possible scenarios:

- **Case 1**: *Hybridization in present layer*
  In other words, even the approximate image of s does not intersect with the next layer. Figure 7.4 is one way to visualize the problem. State s does not belong to

[1]The *Hamming Distance* between two states p and q, denoted by $\mid p-q \mid$, is defined as the number of bit positions where p and q differ.

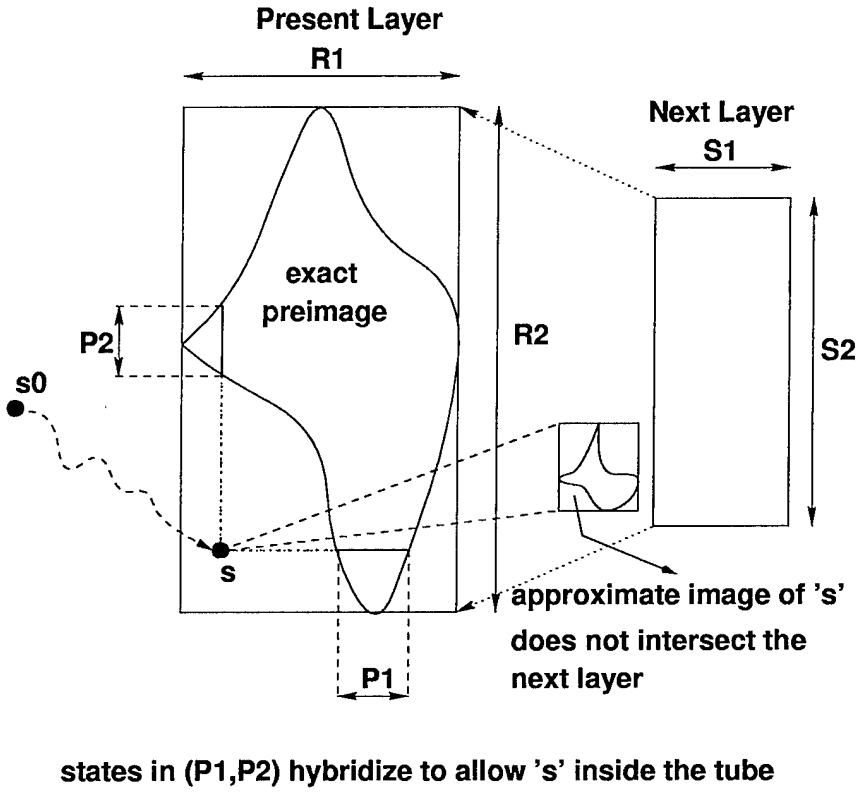


Figure 7.4: Case 1: Hamming distance heuristic to remove bogus states

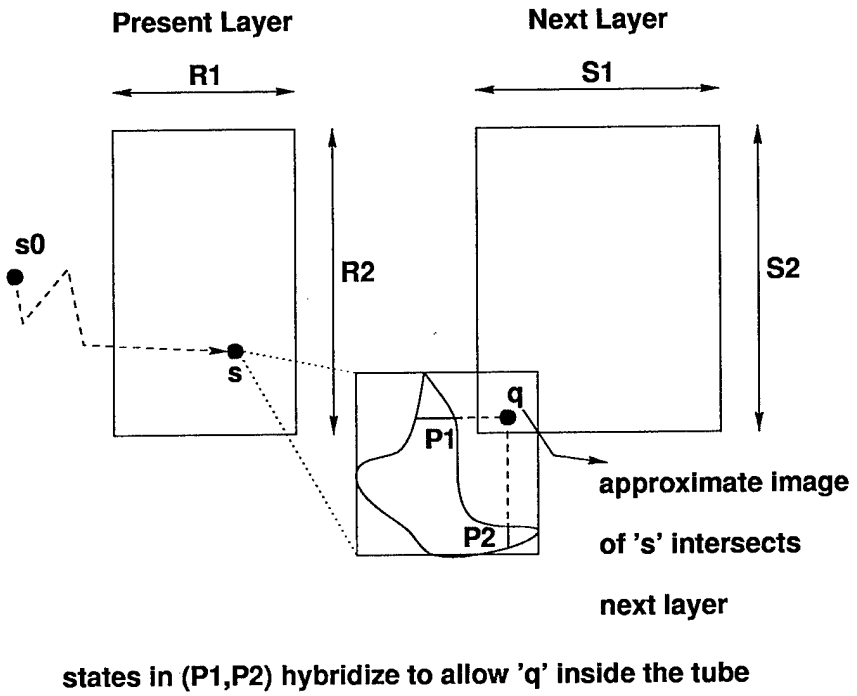


Figure 7.5: Case 2: Hamming distance heuristic to remove bogus states

the exact pre-image of the next layer (or a one step transition would have been possible), but it is included because of the approximation of the pre-image.

Let P_1 represent the set of states in the exact pre-image of $\gamma(S_1, S_2)$ that agree with s on the w_1 bits and P_2 represent the set of states in the exact pre-image of $\gamma(S_1, S_2)$ that agree with s on the w_2 bits. Note that every state in P_1 will hybridize with every state in P_2 to give the state s. The algorithm *ImproveProj* is invoked with the arguments $((P_1, P_2), s, (w_1, w_2))$ to obtain an improved choice of projections.

- **Case 2:** *Hybridization in next layer*
  The approximate image of s intersects with the next layer. Figure 7.5 is one way to visualize the problem.

  Let q be a state in $\gamma(Im_{ap}(s, \mathbf{n}) \sqcap (S_1, S_2))$. Furthermore, let P_1 be the set of states in exact image of s that agree with q on w_1 bits, and let P_2 be the set of states in exact image of s that agree with q on w_2 bits. Every state in P_1 will

hybridize with every state in P_2 to produce the bogus state q. As in the previous case, the algorithm *ImproveProj* is invoked with arguments $((P_1, P_2), q, (w_1, w_2))$ to obtain an improved choice of projections.

7.4.1 Computation of P_1 and P_2

- In **Case 1**, P_1 and P_2 can be computed without computing the exact pre-image $Pre(\gamma(\mathbf{S}), \mathbf{n})$. To compute P_1, all the next state functions are constrained with $\alpha_1(s)$, and then passed to the Pre_{dc} algorithm (which was described in Section 5.3). The other arguments passed to the algorithm Pre_{dc} are **S**, **I**, $\mathbf{x} - w_1$. (where $\mathbf{S} = (S_1, S_2)$ represents the next layer and **I** represents the states generated by previous approximate forward reachability pass). In the cases where the set $\{\mathbf{x} - w_1\}$ is too big and induces many recursive subproblems in Pre_{dc} algorithm (see Section 5.3), the recursive algorithm stops as soon as it finds some state in P_1, and in essence computes an under-approximation of P_1. This entails a possible augmentation of the subsets with a few more bits than the optimal minimum, but our experiments show that this is not a problem. The Hamming Distance causes size increments in the 1-17 range and is well-behaved to ensure that the collection of subsets becomes incrementally coarser. (Analogously P_2 can be computed too).

- In **Case 2**, computing P_1 and P_2 is easy. Since s is a single state, computing the exact image, $Im(s, \mathbf{n})$ is not a problem. P_1 can then be computed by explicitly computing $\alpha_1(q) \wedge Im(s, \mathbf{n})$. (Analogously P_2 can be computed too).

7.4.2 Features of the Hamming Distance Heuristic

1. Since the choice of subsets involves augmenting the various subsets, each pass results in a *coarser* collection of subsets. This guarantees that the results obtained in the next pass with the new choice of subsets are tighter approximations than the results obtained in the earlier pass.

2. Even though more than one correction of the choice of subsets may be required,

it automatically leads to a choice of subsets where transitions can be made from one layer to the next in the approximate tube in one step. Since the layers in the tube are approximations and represent lower bounds on the distance between the initial states and the error states, this finally results in the *shortest possible counterexample.*

3. The method starts with a collection of very small subsets as an initial guess. Thereafter, it automatically finds where the information is getting lost, and iteratively improves the collection of subsets. In contrast to structural methods [11] of choosing the collection of subsets, this is an automatic method of choosing subsets *relative to a property* that needs to be proven.

7.5 Experimental Results

The method was evaluated on the PCI Interface section of the I/O unit from the MAGIC chip, a custom node controller in the Stanford FLASH multiprocessor [45]. Earlier efforts [30] to verify this resulted in many invalid counterexamples because of lack of environment models to model the legal inputs from the PCI bus.

Recently Shimizu *et al.* [62] released a formal specification of the PCI Bus protocol. The monitor-style specification of the protocol by Shimizu *et al.* [62] enabled connecting the I/O section of the MAGIC chip to the monitors (see Figure 7.6). The monitors snoop the transactions on the PCI bus and generate signals $ocorrect_i$, for $1 \leq i \leq 66$, to ensure that only legal inputs from the PCI bus go into the I/O unit. At the same time, the monitors snoop the output signals from the I/O unit to ensure that the outputs obey the PCI bus protocol. These output checking signals form the signals $mcorrect_i$, for $1 \leq i \leq 66$, as in Figure 7.6.

The inputs are constrained to keep all the *ocorrects* high. While doing so, if some $mcorrect_i$ goes to 0, then the PCI unit has violated the PCI specification. Thus, under the assumption of *ocorrects* being true at all times, the tool checks if the validity of the *mcorrects* can be guaranteed. An example of a property from the *mcorrects* is that *irdy* cannot be asserted by the I/O unit in a cycle if the bus was *idle* in the

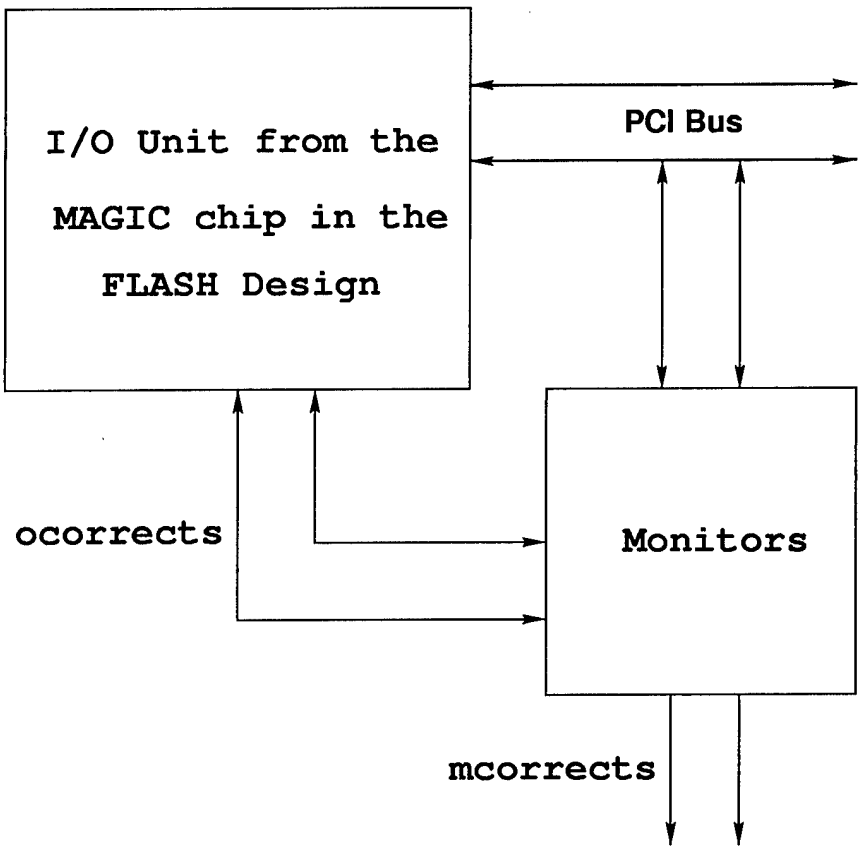


Figure 7.6: PCI design example

previous cycle.

7.5.1 Proving Safety Properties on PCI Interface Unit

The design size of the resulting verification example was 429 state variables and 85 inputs. This verification design example [57], along with the 66 safety properties generated by Shimizu *et al* [62], will soon be publicly available [57].

The initial choice of subsets was based on the heuristic reported in Section 4.5. Since the high level design description (in Verilog) was available, it was possible to identify the various finite state machines by inspection. State variables that encode the state of the same state machine were kept in a single subset. Thereafter, the Hamming distance heuristic improved the choice of subsets. Eventually, the method completed the proof of 64 properties. The remaining two properties could not be proved (nor could counterexamples be generated for them).

The details are in Table 7.1. The column labeled $\#Proj$ refers to the number of subsets. As the size of the individual subsets becomes larger with the Hamming Distance heuristic, some subsets totally subsume other smaller subsets. The totally subsumed subsets are then removed from the collection, which is why the numbers in the column decrease. The column labeled *Avg.* gives the average size of the subsets, and *Max.* reports the size of the biggest subset. The average is a macro level measure of the cost incurred as the subsets become larger, whereas the size of the biggest subset is an indicator of the size of the support set of the potentially biggest BDD in the fix-point routine. (There is a reason why $\#Proj \times Avg. < 429$ (the number of state variables). The 66 *ocorrects* are not included in any of the subsets in w, and instead used as constraints. Further each of the 66 *mcorrects* are included in single sized subsets, which are not counted for the numbers in the $\#Proj$ column.)

The *Size of Tube* (measured in terms of satisfying fraction of the total state space) is an upper bound on the number of states in the final *tube* inside which all counterexamples must lie. As the subsets become coarser, the size of the tube becomes smaller, as expected. The column labeled *Nodes* reports the peak number of BDD nodes alive during the experiment, and *Time* reports the time in seconds. The column *Proved*

reports the number of *mcorrects* proved after that experiment, and *HD Range* gives data on the size increase of some subsets during the experiment. Note that the size of the subsets grew marginally (by 17 bits in the largest case) compared to the size of the final model (which had 429 state variables), even as it enabled the proof of 64 out of 66 properties. It is also interesting that the final choice of projections was able to catch a crucial correlation that enabled a faster time to fix-point.

Table 7.1: Proving safety properties on PCI interface unit

Iter	# Proj	Avg.	Max	Size of Tube	Nodes	Time	Proved	HD Range
1	65	5.06	10	3.646131e-22	298842	2843	11/66	-
2	53	6.35	15	3.207865e-27	307152	3931	11/66	1-8
3	42	8.76	20	9.166265e-30	437237	6496	12/66	4-15
4	41	9.16	29	2.796728e-55	198830	1343	61/66	7-17
5	41	9.17	29	8.739775e-57	128488	977	64/66	1-5
6	41	9.32	29	2.184944e-57	126222	979	64/66	1-3

Counterexample generation

The complete FLASH I/O unit is a very large design with nearly 2400 state variables. To ease the task of verification, initially only the core control portion of the I/O unit was included in the model. Proof of many of the *mcorrects* initially failed and counterexamples for the failed properties were generated. The Hamming Distance heuristic helped in the generation of such counterexamples. Since our search method generates the shortest counterexample, the Hamming Distance heuristic automatically improves the choice of subsets until the number of layers in the tube matches the number of transitions needed to reach the error states. For example, as the choice of subsets improved, the number of layers in the approximation tube increased from 3 to 6 before a valid counterexample (relative to the model) to one of the *mcorrects* was generated.

The designer would inspect the counterexample, and indicate which part of the I/O unit needed to be added to the model to rule out the counterexample. Typically, this occurs because in the process of cutting out parts of the I/O unit, many internal

signals are modeled as non-deterministic inputs in the model. Relevant logic from the I/O unit that drives such signals was added to the model and the verification exercise was repeated. This process of incrementally adding more parts of the I/O unit to the model, as and when it became necessary, resulted in the final model having 429 state variables and 85 inputs. In our experience, the Hamming Distance heuristic is helpful in not only improving the choice of subsets to enable proof of a property, but also in generating actual counterexamples (relative to the model) and by giving information on which other parts of the I/O unit need to be added to the model.

Example of hints provided by the heuristic

The monitor style specification of the PCI bus required that the monitors themselves maintain some internal state depending on the past transactions on the PCI bus. Even the I/O unit of the MAGIC chip has its own internal state depending on the transactions occurring in the PCI bus. The initial choice of subsets left the correspondingly equivalent internal state variables in different subsets. Many of the properties were not provable because of the hybridization effect induced by these equivalent internal state variables not being correlated at all times. The Hamming distance heuristic was able to automatically bring this out, and enabled the proof of many properties.

We conjecture that most or all of the PCI monitor's state is functionally dependent on the state inside the I/O unit implementation. The intuition is that any bookkeeping that the monitors do to ensure the protocol is not violated, must also be done by the I/O implementation to ensure that it obeys the PCI protocol. This is an interesting avenue for future research.

7.5.2 Proving Global Safety Properties on FLASH I/O

The algorithm has also been used to prove some more global properties over FLASH I/O. The whole of FLASH I/O has nearly 2400 state variables. Using the lossless cone-of-influence reduction, the part of the design relevant to the property was extracted. The results are in Table 7.2. *Inv* lists the property being proved. The column under *State* indicates the number of state variables captured after the cone of influence

reduction. P indicates whether the safety property was proved (indicated by Y) or not (indicated by N). *Depth* gives the length of the *shortest* counterexample generated at the end for the cases where the proof failed. The column labeled *Nodes* indicates the peak number of BDD nodes (in millions) during the verification of the property. *Iter* lists the number of iterations of improving the choice of subsets using the Hamming Distance heuristic, before the property was proved or disproved. Finally, *Time* gives the time in seconds for the complete verification exercise.

Table 7.2: Proving global properties on FLASH I/O

Inv	State	P	Depth	Nodes	Iter	Time
AG(p1)	425	Y	-	11.59	2	13152
AG(p2)	233	N	4	4.86	3	11226
AG(p3)	233	N	4	5.28	3	9114
AG(p4)	196	N	7	1.67	3	8083
AG(p5)	196	N	18	1.03	2	5770
AG(p6)	425	Y	-	13.96	1	12745
AG(p7)	196	Y	-	1.22	1	1772
AG(p8)	196	Y	-	1.09	1	1616

The bug exposed by the counterexamples for properties $AG(p2)$ and $AG(p3)$ was later fixed by the designer. It was further proved that the corrected design thereafter indeed satisfied the property (rows for $AG(p7)$ and $AG(p8)$ show results obtained on the corrected design for those properties). (The counterexamples for $AG(p4)$ and $AG(p5)$ were negated by the designer, because they violated some assumptions on the environment under which the design was to operate).

Note that the *Time* column reports the cumulative time spent for the various choice of subsets. This explains why results for properties like $AG(p2)$ which have fewer state variables, still need more time. Since property $p2$ involves 3 iterations of the Hamming Distance heuristic, it means running the algorithm $BackAndForth$ for 3 different choices of subsets. Even though there is a time penalty with repeated traversals over different choices of subsets, the Hamming Distance heuristic has intrinsic value since it is an automatic way of improving the choice of subsets to enable proof of a property.

7.6 Conclusions

The appeal of the ideas in this chapter is the *automatic* failure analysis and *automatic* modification of the subsets to address the failure.

Even though the robustness of the heuristic is evident from the experimental results, there are some avenues for further improvement. Some backtracking could be employed to look for other candidate states in a given layer. This would offset the randomness associated in the present selection of a single state from the exact image to move to the next layer.

Instead of immediately looking for ways to improve the choice of subsets, effort can be put into a more exhaustive search for counterexamples in the current approximation tube. In particular, methods of computing under-approximations of images and pre-images with overlapping projections as the underlying approximation scheme would greatly facilitate this and would fit very well in the overall verification methodology.

Chapter 8

Conclusions

Today's digital systems are designed by a team of designers. Each individual designer has a very detailed understanding of the working of his or her unit and the way it interfaces with other units in the system. The designer's understanding of the other parts of the design is relatively at a much higher level of abstraction. Understanding a design at various levels of abstraction is perhaps the only way a human mind can deal with the tremendous complexity of today's designs. For instance, even though a designer does not have a clear knowledge of the complete state space of the system, he or she may be confident about certain local properties relevant to his or her unit.

It is exactly the same phenomena that makes approximate methods of verification useful. While trying to prove required properties of a design, it helps to look at an abstraction of the design where only the details relevant to the property being checked are included. This helps the verification tool to better manage the complexity of today's designs.

The key idea of this thesis is using a new approximation scheme called *overlapping projections*. Sets of states are represented by an implicit conjunction of small BDDs. The individual BDDs are kept small by restricting their support sets. The scheme allows for using high level information about the circuit structure to guide the approximation. As a result, the scheme is robust enough to handle today's large designs and at the same time retains sufficient information to enable proving correctness properties. The approximation scheme results in tighter approximations compared to

earlier schemes based on disjoint partitions. Overlapping projections allow us to hit intermediate points in the *quality of approximation vs memory space* tradeoff curve, with disjoint partitions on one extreme and exact reachability on the other.

8.1 Key Technical Contributions

In order to make "overlapping projections" a viable approximation scheme, the key technical challenges addressed in this thesis are:

- An efficient *multiple constrain* method for BDDs, that enables us to compute efficiently the image of an implicit conjunction of BDDs with possibly overlapping support, using Boolean function vectors.
- An efficient method based on *domain splitting* along with a number of associated optimizations to compute projections of exact pre-image of an implicit conjunction of BDDs.
- Extracting hidden internal abstractions from the combinational logic, and converting them to *auxiliary variables*, which help further improve the quality of approximation.
- Generating counterexamples from the approximations, and *automatically refining* the approximation in the cases where the tool is unable to give a yes/no answer.

8.2 Key Results

The ideas in this thesis have been evaluated on publicly available benchmark circuits from the ISCAS-89 benchmark suite. Our experiments show *orders of magnitude* improvement in the quality of results obtained when compared with earlier schemes of approximation. The ideas have also been evaluated on a very large realistic design example from the Stanford FLASH Multiprocessor. In particular, the I/O unit of the MAGIC chip in the FLASH Multiprocessor was extensively verified.

Using overlapping projections as the underlying approximation scheme has enabled us to push the limits on the sizes of designs that can be automatically handled by model checking techniques.

8.3 Possible Future Work

Like any thesis, although some answers are provided, many more questions are raised. Even as there is some progress in automatically verifying digital systems, there is still a long way to go before any design is automatically verified. In this section, we suggest some ways in which this work can be further extended.

8.3.1 Better Under-approximations

The weakest link in the present flow is generating counterexamples. The present heuristic for generating counterexamples can be viewed as a naive under-approximation of the reachable state space. When this simple heuristic fails, instead of immediately looking for ways to improve the choice of subsets, more effort can be put on a more exhaustive search for counterexamples in the current approximation tube.

A more general problem is looking for efficient methods to generate better under-approximations of the reachable state space. Throughout this thesis, we have used the high level information on the circuit structure to generate over-approximations (or supersets) of sets of states. To complement this, similar schemes for generating under-approximations (or subsets) of the reachable state space would greatly add to the value of the tool. Existing schemes in the literature for under-approximations do not take into account any high level information about the circuit structure and are hence not very effective for our purposes.

8.3.2 Combining with Other Abstractions

The EDA (Electronic Design Automation) industry today is slowly embracing model checking as a viable commercial product. With the rapid commercialization of model checkers, many of the enhancements and optimizations are often kept secret, making

it hard to do a comprehensive comparative analysis. For example, Cadence Design Systems [9] has a model checking product called *FormalCheck* [10], which has other automatic abstractions (like *localization reduction*) about which little information is publicly available. Even among the research community there has been a lot of interest in abstractions and approximation based methods in the context of model checking. Any model checker could benefit a lot by incorporating all of these techniques into a unified framework. Such a framework, with the high level controls left to the user, would be very empowering, since different techniques work best for different kinds of designs and the user can choose which technique to try depending on the underlying design.

8.3.3 Extension to Liveness Properties

The scope of this thesis is limited to *safety* properties, where the idea is to check that nothing bad happens in any reachable state. For such properties, any falsifying counterexample is always of finite length. Geometrically, a counterexample here represents a path from the initial states to the error states in the state graph. On the other hand, *liveness* properties check that something good eventually happens. For such properties, any falsifying counterexample has infinite length. Geometrically, a counterexample in this case represents a path from the initial states to a strongly connected component in the state graph where each state in the strongly connected component fails to satisfy the eventuality property required.

In order to check for falsifying counterexamples for liveness properties, we need symbolic under-approximate traversal techniques that can look for strongly connected components. The conservative over-approximations paradigm of this thesis is not amenable for checking liveness properties.

8.4 Discussion

The power of formal methods, like model checking, is that they can cover all possible outcomes and hence give absolute guarantees of correctness. But that same power

is also its limitation. The number of possible outcomes is astronomically large for today's large designs, and formal methods do not scale well to deal with such large problem sizes. Therefore, the key for formal methods is to find appropriate approximate methods that can absorb the complexity of today's large designs and at the same time yield useful results. Overlapping projections appears to be an effective approximation scheme to help meet this challenge.

Formal verification with in-built approximation techniques appears to be a very fruitful and promising area for further research. Given the rapid increase in the complexity of today's designs, the traditional simulation based empirical approach of validation will have to be necesarily augmented with approximate formal methods. Given the pressures of early-time-to-market and the reluctance of designers to educate themselves on latest formal verification methods, it is imperative that formal verification tools be automated and easy to use. Thus, efficient approximate methods that automatically refine themselves in the cases where the approximation loses a lot of information, will be the key to ensure that the EDA (Electronic Design Automation) industry adopts them.

Even as this thesis helps further the process of verifying large hardware designs, there is room for improvement and a long way to go before any design is automatically verified.

Bibliography

[1] Abadi, M. and Lamport, L., "The existence of refinement mappings," *Logic in Computer Science (LICS)*, pp. 165-177, July 1988.

[2] Akers, S. B., "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509-516, August 1978.

[3] Balarin, F. and Sangiovanni-Vincentelli, A. L., "An iterative approach to language containment," *Proceedings of Computer Aided Verification (CAV)*, pp. 29-40, 1993.

[4] Bergmann, J. P. and Horowitz, M. A., "Vex - a CAD toolbox," *Proceedings of the Design Automation Conference (DAC)*, pp. 523-528, 1999.

[5] Brayton, R. *et al.*, "VIS: A system for verification and synthesis," *Proceedings of Computer Aided Verification (CAV)*, pp. 428-432, 1996.

[6] Bryant, R. E., "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293-318, September 1992.

[7] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D, L, and Hwang, L. J., "Symbolic model checking: 10^{20} states and beyond," *Logic in Computer Science (LICS)*, pp. 428-439, 1990.

[8] Cabodi, G., Camurati, P., and Quer, S., "Symbolic exploration of large circuits with enhanced forward/backward traversals," *Proceedings of the European Design Automation Conference (EURO-DAC) 1994*, pp. 22-27, 1994.

[9] Cadence Design Systems, `http://www.cadence.com`.

[10] Cadence Design Systems, EDA Solutions Product for Functional/Logic Verification, `http://www.cadence.com/datasheets/formalcheck.html`.

[11] Cho, H., Hachtel, G., Macii, E., Poncino, M., and Somenzi, F., "Automatic state space decomposition for approximate FSM traversal based on circuit analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 12, pp. 1451-1464, December 1996.

[12] Cho, H., Hachtel, G., Macii, E., Pleisser, B., and Somenzi, F., "Algorithms for approximate FSM traversal based on state space decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 12, pp. 1465-1478, December 1996.

[13] Clarke, E. M., Emerson, E. A. and Sistla, A. P., "Automatic verification of finite-state concurrent systems using temporal logic specifications," *Proceedings of* 10^{th} *Annual ACM Symposium on Principles of Programming Languages*, Jan. 1983.

[14] Clarke, E. M., Grumberg, O., and Long, D., "Model checking and abstraction," *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 343-354, 1992.

[15] Clarke, E., Grumberg, O., and Peled, D., "Model checking," *The MIT Press*, 1999.

[16] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H., "Counterexample-guided abstraction refinement," *Proceedings of the Computer Aided Verification (CAV)*, pp. 154-169, July 2000.

[17] Cohn, A., "The notion of proof in hardware verification," *Journal of Automated Reasoning*, vol. 5, no. 2, pp. 127-139, 1989.

[18] Coudert, O., and Madre, J. C., "A unified framework for the formal verification of sequential circuits," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 126-129, 1990.

[19] Cousot, P., and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," *In 4th ACM Symposium Principles of Programming Languages*, pp. 238-252. ACM Press, 1977.

[20] Dams, D., "Abstract interpretation and partition refinement for model checking," *PhD thesis, Technical University of Eindhoven*, 1991.

[21] DeMicheli, G., "Synthesis and optimization of digital circuits," *McGraw-Hill*, New York, 1994.

[22] Dill, D. L., and Wong-Toi, H., "Verification of real-time systems by successive over and under approximation," *Proceedings of Computer Aided Verification (CAV)*, pp. 409-422, 1995.

[23] Eijk, C. A. J., "Formal methods for the verification of digital circuits," *PhD thesis, Eindhoven University of Technology*, 1997.

[24] Eiriksson, A., "The formal design of 1M-gate ASICs," *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pp. 49-63, 1998.

[25] Fallah, F., Devadas, S., and Kuetzer, K., "OCCOM: Efficient computation of observability-based code coverage metrics for functional evaluation," *Proceedings of Design Automation Conference (DAC)*, pp. 152-157, 1998.

[26] Filkorn, T, "Functional extension of symbolic model checking," *Proceedings of Computer Aided Verification (CAV)*, pp. 225-232, 1991.

[27] Fujita, M., Fujisawa, H., and Kawato, N., "Evaluations and improvements of a Boolean comparison program based on binary decision diagrams," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 2-5, 1988.

[28] Geist, D., and Beer, I., "Efficient model checking by automated ordering of transition relation partitions," *Proceedings of Computer Aided Verification (CAV)*, pp. 299-310, 1994.

[29] Govindaraju, G. S., Dill, D. L., Hu, A. J, and Horowitz, M. A., "Approximate reachability with BDDs using overlapping projections," *Proceedings of the Design Automation Conference (DAC)*, pp. 451-456, 1998.

[30] Govindaraju, G. S. and Dill, D. L., "Verification by approximate forward and backward reachability," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 366-370, 1998.

[31] Govindaraju, G. S., Dill, D. L. and Bergmann, J. P., "Improved approximate reachability using auxiliary state variables," *Proceedings of Design Automation Conference (DAC)*, pp. 312-316, 1999.

[32] Govindaraju, G. S. and Dill, D. L., "Approximate symbolic model checking using overlapping projections," *Proceedings of First International Workshop on Symbolic Model Checking (SMC99) at Federated Logic Conference (FLOC)*, pp. 23-33, 1999.

[33] Govindaraju, G. S. and Dill, D. L., "Counterexample-guided choice of projections in approximate symbolic model checking," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, (accepted for publication), November 2000.

[34] Govindaraju, G. S. and Dill, D. L., "Approximate symbolic model checking using overlapping projections," *IEEE Transactions on Computer-Aided design of Integrated Circuits and Systems (T-CAD)*, (under review).

[35] Gupta, A., "Formal hardware verification methods: A survey," *Formal Methods in System Design*, vol. 1, no. 4, pp. 335-383, December 1992.

[36] Hamming, R. W., "Error detecting and correcting codes," *Bell Systems Technical Journal*, vol. 9, pp. 147-160, April 1950.

[37] Hof, D. R., "Intel takes a bullet - and barely breaks stride," *Business Week*, pp. 38-39, January 1995.

[38] Hu, A. J., Dill, D. L., Drexler, A. J., and Yang, C. H., "Higher-Level specification and verification with BDDs," *In Computer Aided Verification (CAV): Fourth International Workshop*, Springer-Verlag, July 1992. Published in 1993 as Lecture Notes in Computer Science, Number 663.

[39] Hu, A. J. and Dill, D. L., "Efficient verification with BDDs using implicitly conjoined invariants," *In Computer Aided Verification (CAV): Fifth International Conference*, Springer-Verlag, 1993. Published in 1993 as Lecture Notes in Computer Science, Number 697.

[40] Hu, A. J., and Dill, D. L., "Reducing BDD size by exploiting functional dependencies," *Proceedings of Design Automation Conference (DAC)*, pp. 266-271, 1993.

[41] Hu, A. J., "Techniques for efficient formal verification using binary decision diagrams," *Ph.D. thesis, Stanford University*, 1996.

[42] Hu, A. J., "Formal hardware verification with BDDs: An introduction," *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 677-682, 1997.

[43] Intel Corporation, `http://www.intel.com`.

[44] Kurshan, R. P., "Timing verification by successive approximation," US Patent US05483470.

[45] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J., "The Stanford FLASH multiprocessor," *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 301-313, April 1994.

[46] Lee, C. Y., "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985-999, July 1959.

[47] Lee, W., Pardo, A., Jang, J., Hachtel, G., and Somenzi, F., "Tearing based automatic abstraction for CTL model checking," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 76-81, 1996.

[48] Long, D. E., As of this writing, a current copy of David Long's BDD package is available via anonymous `ftp` from `emc.cs.cmu.edu`, directory `pub/bdd`, file `bdd.lib.tar.Z`.

[49] Long, D. E., "Model checking, abstraction and compositional reasoning," *Ph.D. thesis, Carnegie Mellon University*, 1993.

[50] Malik, S., Wang, A., Brayton, R. K. and Sangiovanni-Vincentelli, A., "Logic verification using binary decision diagrams in a logic synthesis environment," *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 6-9, 1988.

[51] McMillan, K. L., "Symbolic model checking," *Kluwer Academic Publishers*, 1993.

[52] McMillan, K. L., "A conjunctively decomposed Boolean representation for symbolic model checking," *Proceedings of Computer Aided Verification (CAV)*, pp. 13-25, 1996.

[53] Moon, I, Jang, J, Hachtel G. D., Somenzi, F., Yuan. J., and Pixley, C., "Approximate reachability don't cares for CTL model checking," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 351-358, 1998.

[54] Moon, I, Somenzi, F., Kukula, J. H, and Shiple, T., "Least fixpoint approximations for reachability analysis," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 41-44, 1999.

[55] Moore, G. E., "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114-117, April 19, 1965.

[56] Moore, G. E., "Lithography and the future of Moore's law," *Proceedings of the SPIE*, vol. 2440, pp. 2-17, February 20, 1995.

[57] "PCI verification design example with specification," `http://verify.stanford.edu/pciExample.html`. (to appear).

[58] Pnueli, A., "The temporal logic of programs," *18th IEEE Symposium on Foundations of Computer Science*, pp. 46-57, IEEE Computer Society Press, 1977.

[59] Ravi, K., and Somenzi, F. "High-density reachability analysis," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 154-158, 1995.

[60] Ravi, K., McMillan, K. L., Shiple, T. R., and Somenzi, F., "Approximation and decomposition of binary decision diagrams," *Proceedings of Design Automation Conference (DAC)*, pp. 445-450, 1998.

[61] Ranjan, R. K., Aziz, A., Brayton, R. K., Pleisser, B. and Pixley, C., "Efficient BDD algorithms for FSM synthesis and verification," *Proceedings of IEEE/ACM International Workshop on Logic Synthesis (IWLS)*, May 1995.

[62] Shimizu, K., Dill, D., and Hu, A. J., "Monitor based formal specification of PCI," (to appear).

[63] Thomas, D. E., and Moorby, P. R., "The Verilog hardware description language," *Kluwer Academic Publishers*, 1998.

[64] Thomas, W., "Automata on infinite objects," *Handbook of Theoretical Computer Science*, pp. 165-191, 1990.

[65] Toshiba Corporation, `http://www.toshiba.com`.

[66] Touati, H. J., Savoj, H., Lin, B., Brayton, R. K., and Sangiovanni-Vincentelli, A., "Implicit state enumeration of finite state machines using BDDs," *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 130-133, 1990.

[67] Yang, B, Simmons, R., Bryant, R. E., and Hallaron, D, R., "Optimizing symbolic model checking for constraint-rich models," *Proceedings of Computer Aided Verification (CAV)*, pp. 328-340, 1999.

[68] Yang, C. H., and Dill, D., "Validation with guided search of the state space," *Proceedings of the 35th Design Automation Conference (DAC)*, pp. 599-604, 1998.

[69] Yuan, J., Shen, J., Abraham, J. and Aziz, A., "On combining formal and informal verification," *Proceedings of the Computer Aided Verification (CAV)*, pp. 376-387, 1997.